AFRL-IF-RS-TR-2003-255
Final Technical Report
November 2003

SYSTEM LEVEL APPLICATIONS OF ADAPTIVE COMPUTING (SLAAC)

University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J450

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** NOVEMBER 2003	**3. REPORT TYPE AND DATES COVERED** Final Jul 97 – May 03

4. TITLE AND SUBTITLE
SYSTEM LEVEL APPLICATIONS OF ADAPTIVE COMPUTING (SLAAC)

5. FUNDING NUMBERS
C - F30602-97-2-0220
PE - 62301E
PR - D002
TA - G2
WU - P8

6. AUTHOR(S)
Robert Parker, Brian Schott, Matt French, Ronald Riley, Brad Hutchings, Brent Nelson, Michael Wirthlin, Rick Pancoast, Maya Gokhale, Kevin McCabe, Brian Bray, John Villasenor, Peter Athanas, Mark Jones, and SLAAC Research Group

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina Del Rey California 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTC
26 Electronic Parkway
Rome New York 13441-4514

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-255

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Wilmar Sifre/IFTC/(315) 330-2075/ Wilmar.Sifre@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 Words)*

SLAAC assembled a multidisciplinary team with expertise in signal processing, design tools and adaptive architectures to develop distributed adaptive computing systems (ACS) and methodologies for Defense processing. SLAAC was organized around challenge applications that collectively guided the research, including: Synthetic Aperture Radar/Automatic Target Recognition (SAR/ATR), SONAR Beamforming, Wide-Band RF Receiver Processing Hyperspectral Image Processing, Electronic Counter Measures (ECM), and Infrared ATR. The challenges were selected to stress ACS systems in dimensions of computational density, memory bandwidth and latency, I/O bandwidth, and scalability. SLAAC had technology transitions in SAR, SONAR and ECM.

Core research activities were in scalable architectures, adaptive hardware, runtime software, module generators, and domain compilers. The research led to a hardware-accelerated cluster computing programming model and ACS API runtime system. SLAAC produced nearly 100 PCI and VME ACS accelerator boards, spanning three technology generations. The hardware is licensed commercially. Tool development included Streams-C and extensions JHDL. Streams-C is a high-level stream-oriented C compiler for hardware and software. JHDL is a hardware design language based in Java that includes tools for creating, simulating, executing, and debugging FPGA hardware designs. SLAAC contributed a large library of computation modules to JHDL and extended it to be a domain-specific compiler.

14. SUBJECT TERMS
Adaptive Computing Systems, SONAR, RADAR, FPGA, JHDL, Streams-C, Electronic Counter Measures, Automatic Target Recognition, SAR/ATR, SONAR Beamforming, Wide-Band RF Receiver Processing, Hyperspectral Image Processing

15. NUMBER OF PAGES
621

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 EXECUTIVE SUMMARY

SYSTEM LEVEL APPLICATIONS OF ADAPTIVE COMPUTING (SLAAC)
Robert Parker, Brian Schott
University of Southern California, Information Sciences Institute

The primary focus of the System Level Applications of Adaptive Computing project in DARPA Adaptive Computing Systems (ACS) was to assemble a multidisciplinary team with application domain expertise, design and software expertise, and adaptive computing systems expertise to develop and demonstrate system-level methodologies for adaptive computing systems applied to Defense heterogeneous processing. Other individual ACS efforts had focused on specific tools, hardware components, and low-level designs and had not attempted the complex challenge of designing large-scale systems with hardware whose architecture is malleable at the gate level. The SLAAC effort proposed to: 1) create a design methodology, open architecture definition, and runtime environment based on emerging embedded high performance computing and ACS components; 2) create a generic reference implementation for the research community to accelerate technology demonstration and insertion across several processing domains; and 3) extend the reference implementation particularly in the SAR/ATR domain to demonstrate 500x improvement in performance density.

1.1 SLAAC Technology Development

The core research activities on the SLAAC project outside of application development were in the areas of scalable systems architectures, adaptive computing hardware, runtime software, module generators, and domain compilers. The hardware and software technology development on the SLAAC project was motivated by an impressive number of challenge applications with diverse requirements such as computational density, memory bandwidth, memory access latency, I/O bandwidth, end-to-end processing latency, computational variability, and system scalability. Overall, the adaptive computing systems and tools developed under the SLAAC project have been widely distributed, are openly available and in wide use within the community at universities, government laboratories, and commercial companies. Specific technologies developed under the SLAAC project include:

1.1.1 <u>ACS Runtime System</u>

One of the dilemmas of architecting a general purpose adaptive computing system for a variety of challenge applications is achieving a system that can truly scale to the real problem. To tackle this issue, the SLAAC team developed a scalable systems architecture concept based on a high-speed network of hosts, nodes, and channels. The physical implementation of this architecture was borrowed from cluster computing. It consisted of FPGA boards installed in Linux workstations with Myrinet high-speed networking. This reference system was called the Research Reference Platform. The team developed a distributed control library interface and runtime system, the ACS API, on top of the Message Passing Interface (MPI). An embedded version of this architecture, called the Deployable Reference Platform, was developed using a VME-based multicomputer in place of a workstation cluster. The API has been ported to all SLAAC boards and many other commercial and research boards being used in the ACS community. The source code and specification of the ACS API are available without restriction at www.ccm.ece.vt.edu.

1.1.2 SLAAC-1 PCI Accelerator

SLAAC-1 was a PCI accelerator board developed in 1999 for the SLAAC Research Reference Platform. It had one Xilinx XC4085 and two XC40150s organized in a “systolic ring and crossbar” architecture. It had approximately 750,000 user programmable gates. The board also had ten banks of 256K x 18-bit 100MHz SRAM. One XC4062 implemented the PCI interface. SLAAC-1 was the template from which all subsequent architectures were derived. The three user FPGAs and ten memories were synchronous and had a programmable clock controlled by the interface chip. SLAAC-1 had an onboard configuration cache for user designs. Streaming data entered from FIFOs in the interface chip. The SRAMs were memory mapped in the host address space and could be preemptively accessed from the host. Eleven production boards were built and delivered to members of the SLAAC team.

1.1.3 SLAAC-2 Dual VME Accelerator

SLAAC-2 was also developed in 1999 for the SLAAC Deployable Reference Platform. It was a mezzanine board designed to plug into a modified CSPI M2621 VME Multicomputer, which has two PowerPC 603s running VxWorks. SLAAC-2 was divided into two nodes, one per processor. The hardware architecture of each node was identical to SLAAC-1. In fact, the FPGAs were binary compatible to facilitate application development in an inexpensive workstation environment and only move to the VME platform during final application integration. SLAAC-2 had a two Xilinx XC4085s and four XC40150s, about 1.5M logic gates. It also had 20 banks of 256K x 18-bit 100MHz SRAMs. Two XC4085 interface chips interfaced directly to the PowerPC busses of the CSPI board. Only three SLAAC-2 boards were produced. One was delivered to Lockheed Martin Government Electronic Systems in Morrestown, NJ. A second was delivered to Integrated Sensors Inc. for RT-Express development. The third remains at ISI.

1.1.4 SLAAC-1V Virtex PCI Accelerator

SLAAC-1V was developed in 2000 to explore the capabilities of new Xilinx Virtex devices, particularly fast partial runtime reconfiguration. The PCI board had three Virtex-1000 FPGAs, approximately 3M logic gates, in the “systolic ring and crossbar” architecture. SLAAC-1V was designed to be source-code compatible with SLAAC-1 designs. The larger devices supported wider memories, ten banks of 256K x 36-bit 200MHz SRAM. Instead of a separate interface chip, SLAAC-1V had the 64-bit 66MHz PCI interface in X0. A small Virex-100 was used as a configuration controller. Approximately 50 SLAAC-1V boards were produced; some of which were externally funded.

1.1.5 Osiris Virtex-II PCI Accelerator

Osiris was developed in 2001 to exploit the computational density of Xilinx Virtex-II. Instead of three chips, this full-sized PCI card had one large Virtex-II 6000 FPGA for approximately 6M logic gates. The board had considerably more onboard memory, ten banks of 512K x 36-bit 250MHz SRAM and two SODIMM sockets with up to 512MB of PC133 SDRAM. Osiris added a standard PCI Mezzanine Card (PMC) connector for external I/O. The Virtex-II 1000 interface chip has a bridge between two PCI busses; the left-hand interface connects to the host and the right-hand interface connects to PMC. Approximately 30 Osiris boards were produced. Osiris was licensed to Atlantic Coast Telesys and they are now being produced commercially.

1.1.6 PipeRench PMC Board

Part of the SLAAC project charter was to assist other DARPA ACS projects demonstrate their technology. We produced a PMC daughter card for Carnegie Mellon University. It had a Virtex-II 1000, two banks of SRAM, and two PipeRench sockets. Ten PipeRench PMC boards were produced and delivered to CMU for their demonstrations.

1.2 SLAAC Challenge Applications

The SLAAC team effort was organized around a number Defense challenge applications that collectively guided the innovative hardware and software development. These application challenges were selected to stress the ACS systems under development in a number of dimensions, such as computational density, memory bandwidth, memory access latency, I/O bandwidth, end-to-end processing latency, computational variability, and system scalability. Not all of the application challenges would necessarily be solved by ACS technology, but the diversity of the problem space attempted would lead to more general purpose adaptive computing systems. Specifically, the SLAAC application challenges were:

1.2.1 SAR/ATR Application Domain Challenge

The challenge was to use the latest ACS technology to accelerate EHPC based partial signature contamination algorithms for SAR ATR for Joint STARS. The goal is 500X the density of a scaled 1996 baseline system that would support a data rate of two megapixels per second for 30 target configurations. Early in the SLAAC project, the team completed performance analysis studies on a number of key components of the Sandia algorithm suite.

One of the earliest algorithms we looked at was the first part of Focus Of Attention (FOA), which is adaptive quantization. FOA processing was specified in an image morphology language called CP4L originally developed for the CYTO image processor. Sandia had a requirement to be able to change the morphology scripts without needing a hardware designer. The BYU team developed a compiler tool on top of JHDL that generated FPGA designs from CP4L. The average throughput of the resulting system is 46 megapixels, which is 10X comparable workstation performance. The FOA system was delivered to Sandia.

Another algorithm component we looked at early in the SLAAC project was Second Level Detection (SLD), but this was soon replaced at Sandia by a more robust algorithm called Contamination Distribution Indexer (CDI). CDI extends the capabilities of the Sandia system by supporting camouflage, concealment, and deception scenarios. CDI was considerably more computationally intensive. The goal was to develop an ACS-based system that was also 10X the performance of an embedded VME multicomputer. We were able to achieve 40X the throughput of a single embedded PowerPC (10X a Quad PowerPC Multicomputer).

The final result of the end-to-end system test in the laboratory on a Sun Ultra 60 with SLAAC-1V at Sandia was 1.09 megapixels per second. SLAAC-1V compatibility problems with the rack-mounted Sun Blade 1000 prevented us from testing on the embedded system, but adding the software performance on Blade 1000 for the unaccelerated Sandia modules to hardware/software performance of CDI and FOA on Ultra 60 with SLAAC-1V yields a performance of 2.5 megapixels per second. This exceeds the SAR/ATR performance goal.

1.2.2 SONAR Application Domain Challenge

The SONAR challenge problem was selected because conventional wisdom at the time indicated that SONAR was beyond the capabilities of FPGAs. It violated three of the marker characteristics of successful FPGA applications: 1) small data elements, usually 8-bit or smaller, 2) simple arithmetic, mainly Boolean operations and additions/subtractions, and 3) minimal control logic. Work on this challenge started with a number of feasibility studies on algorithms provided by the Naval Undersea Warfare Center (NUWC). It then moved to ACS kernel development for the matched field algorithm developed at NUWC. We delivered a sonar design on SLAAC-1 to NUWC in June 2000, hoping for a ride on the next available sea trial. This first-generation SLAAC-1 platform running at 50MHz delivered the equivalent of between 4 and 5 billion arithmetic computations per second as the algorithm would run on a cluster of workstations. NUWC has been using this testbed since for computations on synthetic and recorded sea data in its testbed. In 2001, some analysis was started in the SONAR area looking at floating-point algorithms and modules. However, this work was taken over by the DARPA Robust Passive Sonar program. As a final technology transition, SLAAC delivered 6 Virtex-II Osiris boards to NUWC in April 2002 for algorithm development under the RPS program.

1.2.3 AEGIS Electronic Counter Measures Challenge

The AEGIS Electronic Countermeasures Assessment (ECMA) Signal Processing Subsystem provides for detection and analysis of countermeasures and jamming signals in the AEGIS AN/SPY-1 shipboard phased array radar system. The Advanced ECMA Processor was of interest because it provided a meaningful test case for the applicability of the Adaptive Computing technology to a real fleet problem where integration issues dominate. As a result of efforts initiated in 1995 at Lockheed Martin Government Electronic Systems (LMGES) to upgrade the existing AN/SPY-1 ECMA processor, the U.S. Navy AEGIS Program Office was interested in leveraging ongoing DARPA programs to upgrade and simplify legacy processors such as ECMA. The ECMA equipment frame consists of approximately 35 different types of application specific electronic modules, filling an entire six-foot high, nineteen-inch wide equipment frame. The hardwired architecture is fixed, not programmable, not flexible from a reconfiguration standpoint, and is not scaleable as requirements change. In the ECMA challenge, the goal was to achieve a volumetric reduction with identical functionality. LMGES identified fourteen major modules for the SLAAC developers to target. In 1999, all fourteen modules were implemented on a SLAAC-2 VME board achieving a 95% volume reduction with identical waveform outputs. The system was demonstrated to the Navy in 2000 and LMGES has successfully transitioned ACS technology to the Navy PEO TSC TI and production programs.

1.2.4 Wide-Band RF Application Domain Challenge

Los Alamos National Laboratory (LANL) Nonproliferation and International Security Division (NIS) provides technology to a wide range of DoE and DoD agencies. Their role in SLAAC has been to direct and validate DARPA technology via technology insertions into ongoing DoD programs. The Wide-Band RF Challenge application was selected to look at ACS technology suitability for high streaming bandwidth applications usually solved by ASICs. The small production quantities and desire to specialize make digital receivers an attractive application for FPGAs. LANL identified the Digital Receiver Component Library package as an appropriate target for SLAAC development. This package has a set of common signal processing ASIC functions. The challenge goal is to accelerate a channelized detector to achieve 100 megasamples per second processing. Six component modules were implemented including: 32 Channel Polyphase Filter Bank, 32 channel FFT, Statistical Detection Algorithm, Direct Digital Frequency Synthesizer, Adaptive Decimation, and Log CORDIC. These modules are representative of those used in the Bellatrix airborne reconnaissance program, the Cibola Ground Demo program (space based digital radio), and Martes/Midas2K analyst workstation. The Digital Receiver Library was implemented as planned. This was an enabling technology for the Capella, Bellatrix and Cibola wideband RF programs at LANL and is the foundation for several programs on the near horizon. Capella led the way with early engineering studies but the first true delivery was by Bellatrix to the Air Force with a successful airborne flight demonstration in July 2000. The success of these programs depended on the ability of reprogrammable ACS systems to support iterative development from early proof of concept designs, to detailed performance-tuned implementations, to in-system design upgrades, at a much lower risk than a comparable ASIC solutions.

1.2.5 Hyperspectral Imagery Application Domain Challenge

Los Alamos National Laboratory (LANL) Nonproliferation and International Security Division (NIS) provides technology to a wide range of DoE and DoD agencies. An important aspect of their research is looking at feature-recognition techniques for multi-dimensional image sensor projects. The goal of the Hyperspectral Imagery Challenge was to examine how ACS systems could be applied to large image data cubes. There were several application studies in this class.

The Rapid Feature Identification Project (RFIP) and Accelerated Image Processor using Machine Learning (POOKA) application demonstrated an evolvable hardware approach to feature recognition for multispectral imagery. After training, the POOKA system demonstrated two orders of magnitude speedup. The POOKA system was deployed at the National Imagery and Mapping government agency (NIMA) for beta testing since March of this year. The system has been successfully applied to practical broad area search problems and attracted the attention of a second government customer. LANL expects to deliver a second system in June of this year.

The HIRIS challenge can be characterized as having two separate technical thrusts – precise chemical analysis and chemical detection. The HIRIS Hyperspectral sensor collects data as a sequence of 128x128 broadband IR images taken by an interferometer for a range of lengths of its reference leg. Converting this data to an image of high spectral resolution requires computing the Fourier transform/power spectrum for ~8k samples for each of the 128x128 pixels. The spectra of each pixel must be calibrated independently. The densities of a number of gases are then estimated from this spectrum. Storage capacity places severe constraints on the number of raw image cubes that can be collected. The detection challenge is performing the entire processing chain in real time, ~3 seconds, with sufficient resolution to determine if a collected image is worth archiving. In a programmatic sense HIRIS has been a disappointment due to factors outside the scope of our ACS effort. There has been a great state of uncertainty in the hyperspectral programs at the national level. The result was instability in the programs we targeted such that the goals and problems to be addressed or even the existence of a program changed faster than we could respond. Despite this we demonstrated our ability to accelerate both statistical and explicit image processing and thus we are well prepared for the needs of the next generation hyperspectral camera.

1.2.6 IR/ATR Challenge Problem

The IR-ATR (Infrared Automatic Target Recognition) algorithm was developed at Night Vision Laboratories (NVL) under the ATR Relational Template Matching program (ARTM) to enable a real-time image cueing system for M1 tank operators driving at night. The system needed to be fast enough to process large, 180° or more views of high-resolution image data, with refresh rates of no less than 10 Hz. The system also had to be small enough to fit into the tight space of a tank, in this case, a rugged equipment enclosure with only 2 VME slots available. The algorithm had the form of a 6-level decision tree, with the top level (Round 0) representing about 90% of the computation. For the SLAAC effort, the goal was to see if an ACS accelerated embedded multicomputer cluster (the Deployable Reference Platform) could succeed in a very constrained real-time environment. Our partitioning of the problem specified a SLAAC-2 VME board in one slot entirely for Round 0 and the second slot occupied with a COTS PowerPC board for the remaining rounds. The UCLA team used a bitfile compatible SLAAC-1 board and developed a hardware algorithm for implementing the sparse template. The UCLA approach was to infuse all Round 0 templates into a single adder tree. UCLA was able to demonstrate a Round 0 design on SLAAC-1 that met a clock rate that could achieve the 100ms window on the end-to-end system. However, delays in firmware integration for SLAAC-2 caused us to miss our short demonstration window. NVL had subsequently demonstrated this application using more traditional commercial FPGA hardware.

1.2.7 PAPMU Challenge Problem

BAE Systems is conducting low-power research under the DARPA funded CSPAD and AMPS programs and identified a suite of processing, similar to that of the SLAAC Wide-Band RF application, appropriate for ACS technology that would complete an end-to-end low power system. The goal of this challenge area is to reduce the power consumed by the parameter measuring unit ACS implementation by 10x. The focus of this research is to reduce the power utilized by intelligent FPGA algorithm mapping, device utilization, and component placement. ISI developed several techniques to reduce the dependency on high power components such as multipliers and long line interconnects and developed a library of several FIR filters incorporating these techniques. A hardware demonstration, on the commercialized Osiris boards, of the baseline algorithm versus the optimized algorithm using the low-power filter library was performed before DARPA and AFRL officials and showed a 54.5 times improvement in operations per Watt. This technology has been transitioned to a NASA effort under which ISI is developing CAD tools for better analyzing and optimizing FPGA power performance.

2 FINAL STATUS REPORT ON ADAPTIVE COMPUTING SYSTEM ARCHITECTURES

Mr. Brian Schott
University of Southern California, Information Sciences Institute

2.1 Introduction

The mission of the System Level Applications of Adaptive Computing (SLAAC) project is to: 1) define an open, distributed, scalable, adaptive computing systems architecture; 2) design, develop, and evolve scalable reference platform implementations of this architecture; and 3) validate the approach by deploying technology in multiple defense application domains. In the context of this research, adaptive computing systems (ACS) refer to systems that reconfigure their logic and/or data paths in response to dynamic application requirements. The SLAAC effort brings to bear a strong multidisciplinary team of researchers with application domain expertise, design and software expertise, and adaptive-computing expertise with team members distributed among several universities, government labs, and defense contractors.

Some of the application challenges targeted by SLAAC such as SAR/ATR are expected to scale to tens of VME slots in order to satisfy real-world processing requirements. Transitioning from a small proof of concept demonstration to a large real-world application is often overlooked in ACS research. For example, in SLAAC, a majority of the selected demonstrations require strict VME environments yet most of the algorithm mapping is being done in university research labs. Replicating these VME environments in the university labs would be prohibitively expensive because of the cost of chassis, single-board computers, and real-time software development tools. In addition, there is a learning curve associated with programming and debugging embedded systems that make them a poor choice for an application development platform for graduate students.

Conversely, allowing the university partners to develop their applications on commercial ACS boards in a PC environment is equally impractical because no path exists to transition to the deployed environment. There is little commonality in ACS hardware architectures and software APIs. The task of porting to a VME platform becomes as difficult as performing the original algorithm development. Finally, the applications targeted by SLAAC are estimated to be larger than a single PCI-based ACS board. So, scalability in addition to portability is another issue in transitioning from the research lab to a field environment.

The SLAAC team approach to these issues is to make platform-independent distributed ACS systems that are scalable and source-code compatible to allow application development in the research lab transition quickly to deployed systems. Achieving this goal requires closely coordinated hardware and software development in next-generation ACS accelerators, module generators and other design tools, runtime control libraries and APIs, and algorithm mapping.

2.2 System Architecture

The SLAAC distributed system architecture view defines a *system* as a collection of *hosts, nodes,* and *channels*. A *host* is an application process responsible for allocating nodes and setting up channels in the system. A host sends control commands over the network to nodes in the system. A *node* is a computational device such as an adaptive computing board. Nodes are logically numbered during system creation. *Channels* are logical FIFO queues that interconnect nodes and hosts. Channels have *endpoints*, which are typically physical FIFOs on an adaptive computing board and buffer queues in a host process. The underlying runtime system is responsible for moving channel data across the network. From the application programmer's perspective, channels are autonomous data streams. Figure 2-1 shows a conceptual view of two ACS systems in a network.

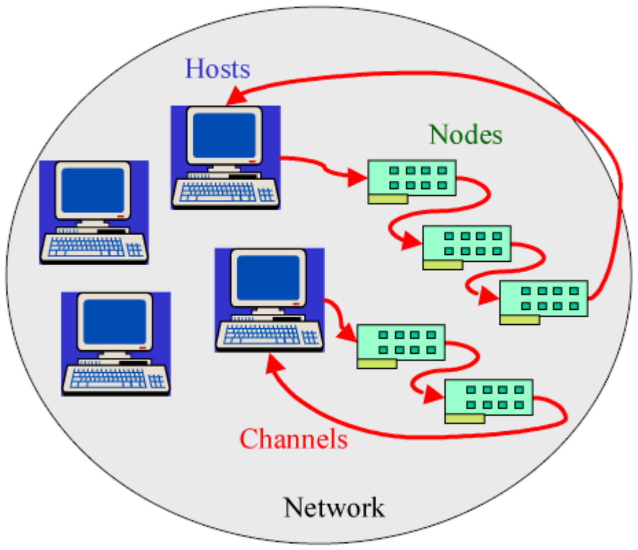


Figure 2-1 ACS System View

This conceptual ACS system view of hosts, nodes and channels presents a programming model that has several distinct advantages for ACS developers. First, the notion of streaming data through an FPGA array using FIFOs is a familiar technique to ACS application designers. Chaining multiple ACS board designs together using channels is a fairly simple extension to this idea. The ACS programming model also supports the alternate technique of reading and writing to the on-board memories of FPGA boards for communication. A logical node number argument is added to read and write functions to access different boards in the system. A second advantage of the ACS programming model is that it encourages application designers to decompose their problem into multiple ACS board-sized modules. These modules can be individually designed and tested on single ACS boards before being connected together with channels, thereby improving code modularity and reuse. Third, this approach makes it easier to exploit coarse-grained parallelism in an application. By replicating board-sized modules and distributing the compute load (such as distributing templates or partitioning images), an application can often scale to the number of ACS boards available in the system and gain a coarse-grained parallelism advantage.

In order to support the SLAAC programming model, the SLAAC team has defined an API for controlling an ACS system from one or more host processes. The ACS system API has system creation functions for node allocation, memory access functions for reading and writing a node's local memories, channel functions for streaming data through the system, and convenience functions for common ACS board control routines such as configuration, readback, and clock programming. The ACS library is implemented in C++ with a C front-end. The ACS system

API borrows heavily from the Message Passing Interface (MPI) standard and is intended to operate easily as middleware on top of MPI. The ACS system API can cooperate seamlessly in an existing MPI application using a private communicator or entirely encapsulate the MPI calls to provide a simplified programming interface for controlling the ACS system from a single host process.

Although the SLAAC programming model and ACS system API definitions attempt to make very few assumptions about the underlying system hardware (this is essential in developing portable APIs), the SLAAC team has carefully constructed two hardware reference implementations of distributed ACS systems. The Research Reference Platform (RRP) is a network of ACS-accelerated workstations. The RRP is appropriate for application development in a university lab environment or processing environments where physical form factor is of less concern. The second hardware reference implementation being created by the SLAAC team is called the Deployable Reference Platform (DRP). The DRP is a VME-based system appropriate for real world processing in existing defense systems where compute density and physical form factor are the primary concerns. Even though they have different operating systems and hardware environments, the SLAAC team seeks to demonstrate that it is possible to build RRP and DRP systems that are source code compatible.

2.2.1 Research Reference Platform

The cluster computing community uses inexpensive workstations and high-speed networks to build high performance parallel systems. The Research Reference Platform (RRP) introduces ACS boards into this workstation cluster and provides the ACS system API to simplify application programming. The RRP has the advantage of being an inexpensive readily available platform for ACS development that tracks advances in commercial high speed networking, workstations, and adaptive computing. In hardware terms, the RRP is a naturally occurring phenomenon throughout the ACS community. As researchers collect ACS boards in their labs, they are routinely inserted into network environments for the purpose of sharing the equipment. The ACS system API unifies the programming environment for these systems.

The traditional configurable computing approach to solving large applications was to assemble massive systolic arrays of FPGAs as in the case of the SPLASH 2 machine. However, the SLAAC team proposes that an RRP can be used effectively to approximate these systems and solve the same applications with greater flexibility and at much lower cost. A good example of an RRP is the Tower of Power (ToP) at Virginia Tech. The ToP has sixteen PCs each equipped with a WildForce™ FPGA accelerator board and a Myricom System Area Network (SAN) interface card connected to a sixteen port Myrinet™ switch. A Wildforce board has four processing elements consisting of a Xilinx XC4062XLA FPGA and a local 32-bit SRAM. Chaining the input FIFOs to the output FIFOs of four of these boards using high-speed network channels is a good approximation of a single SPLASH 2 systolic array board. SPLASH 2 contains 16 processing elements, each is a Xilinx XC4010 FPGA and a local 32-bit SRAM.

Much of the SLAAC research in the RRP domain is concentrating on the ACS system layer API and runtime library. The intention of the RRP is to allow a variety of ACS boards in a network cluster to be used interchangeably without having to make significant changes to the host application code. The different control APIs of ACS boards are hidden under a single ACS system layer API. The ACS runtime system is currently available in Linux and Windows NT™

implementations and supports ACS boards such as WildForce™, WildOne™, and the SLAAC-1 board being developed by the SLAAC hardware team.

2.2.2 Deployable Reference Platform

The Deployable Reference Platform (DRP) is a demonstration that SLAAC can achieve the same distributed ACS system architecture in a field-friendly platform. Unlike the RRP, applications for the DRP are very particular about compute density and measure performance normalized by power, weight, and volume. An added challenge is that the DRP platform is required to be source code compatible with an RRP platform so that university researchers can develop applications and easily transition them to deployed systems. Since the RRP is an ACS accelerated network cluster, the SLAAC team approached the problem by looking for a VME-based cluster computer to accelerate.

The commercial CSPI M2641 has four 300MHz PowerPC™ 603r processors connected to a Myrinet 1.2Gb/sec System Area Network (SAN) network. The M2641 has an integrated 8-port switch and supports network connections from both the front-panel and the P0 VME backplane row for cable-free networking. The SLAAC team collaborated with CSPI to modify the baseboard half of their M2641 multicomputer product. The mezzanine board of the M2641 contains two PowerPCs and this mezzanine was discarded to provide room for the SLAAC-2 ACS accelerator board. The SLAAC-2 board provides independent FPGA acceleration to both of the remaining PowerPCs. Two PowerPC bus connectors were added for this purpose. This board is now commercially available as the M2621S.

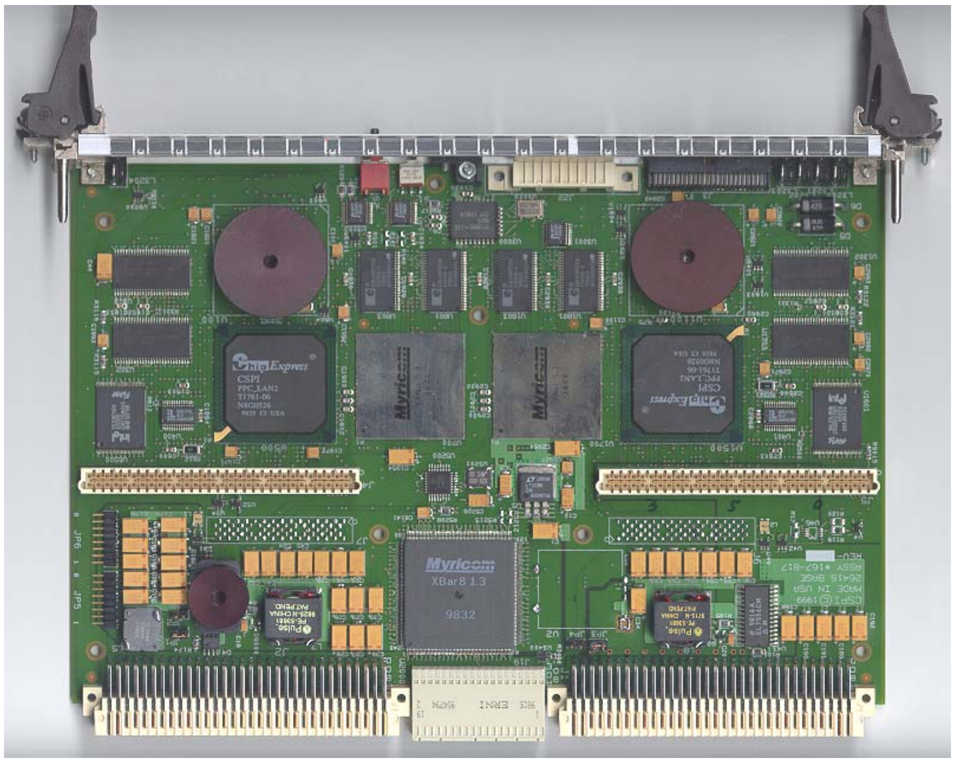


Figure 2-2 M2621S Photo

Figure 2-2 shows the M2621S carrier that has been modified for SLAAC-2. The black circular heat-syncs conceal the PowerPC™ processors in the upper-left and upper-right corners. The BGA packages near the center of the board from left to right are 1) an ASIC interfacing the 2) Myricom LANai™ network processor to the PowerPC bus, and the 3) LANai processor and 4) ASIC FPGA interface for the second PowerPC node. The M2621S board has been modified for the SLAAC project to include two 120-pin PowerPC bus connectors visible in the SLAAC-2 photo. Two additional 60-pin SAN connectors are also available. However, they are unpopulated in this photograph because SLAAC-2 does not use them.

From a software perspective, the M2621S carrier and SLAAC-2 board represent two ACS-accelerated PCs in a single VME slot. The M2621S board uses VxWorks™, a Unix-like operating system for real-time embedded systems. An efficient implementation of MPI is available for the M2641, which allowed the ACS system layer API to be easily ported to the DRP. This means that the DRP is source compatible with the RRP with respect to the C host control code. The SLAAC team has approached the other compatibility issue – the ACS accelerator – by making sure the SLAAC-1 and SLAAC-2 ACS boards share a common hardware architecture. This architecture is discussed in Section 2.3.

2.3 Hardware Architectures

The SLAAC hardware architecture is an attached ACS accelerator comprised of processing elements containing FPGAs and fast local memories. The basic concept is comparable to predecessor reconfigurable computer architectures such as SPLASH 2 and Wildforce™. The intent is to incorporate the most successful features from previous reconfigurable computing architectures and utilize the largest FPGA devices currently available. The SLAAC project produced five distinct boards: SLAAC-1, SLAAC-2, SLAAC-1V, Osiris, and PipeRench. They are discussed in detail in the next sections.

2.3.1 SLAAC-1 (PCI)

As shown in Figure 2-3, the SLAAC-1 architecture is partitioned into a single interface FPGA (labeled ‘IF’) and three user-programmable FPGAs (labeled ‘X0’, ‘X1’, and ‘X2’). The IF chip is configured at power-up to act as a stable bridge to the host system bus. It provides configuration, clock, and control logic for the user FPGAs. The attached host is responsible for actually programming the user FPGAs and controlling the board. SLAAC-1 supports DMA to transport data to and from host memory. A clock generator and FIFOs implemented within IF allow the user FPGAs to operate at the optimal frequency for the application design.

2.3.1.1 Data Paths

One of the goals of the SLAAC-1 architecture was to design an FPGA accelerator assuming a 64-bit data word. Since fast 64-bit system busses are becoming more commonplace in commodity PCs, it was felt that a 64-bit data word was necessary to keep up with modern I/O rates. A 64-bit word is also a more natural atomic data element for these wider processors and even/odd word alignment issues of a 32-bit FPGA system would cause additional complexity in user FPGA designs. Consequently, the two bi-directional 72-bit “FIFO” connections between IF and X0 permit the user FPGAs to produce and consume a 64-bit data word in a single clock cycle. The three user-programmable FPGAs are organized in a ring structure. X0 acts as the control element for managing user data flow, thus enabling X1 and X2 to focus on computation.

The ring path (X0→X1→X2→X0) is also 72 bits wide so that an 8-bit tag can be associated with each 64-bit data word. The individual pin directions on the ring connections are user-controlled; this architecture could just as easily support one 36-bit clockwise ring, and one 36-bit counterclockwise ring. The "crossbar" connecting X0, X1, and X2 together is a common 72-bit bus. The user also controls the direction of individual pins of this crossbar. Six additional handshake lines not shown (two each from X0 to X1, from X1 to X2, and from X0 to X2) permit crossbar arbitration without requiring unique configurations in X1 and X2.

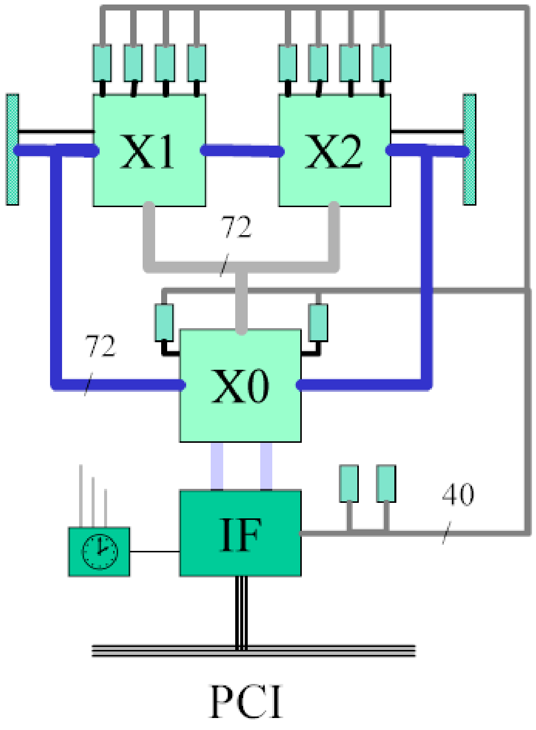


Figure 2-3 SLAAC-1 Architecture Diagram

2.3.1.2 Processing Elements

The SLAAC processing elements X1 and X2 each consist of one Xilinx XC40150XV-09 FPGA and four 256Kx18bit synchronous SRAMs. The Xilinx 40150 contains a 72x72 array of CLBs for 300K equivalent logic gates supporting clock speeds up to 100MHz. The SRAMs feature zero-bus turnaround permitting a read or write every cycle; no idle cycles are required for write after read with the only tradeoff being that write data is pipelined one cycle. Each PE has two 72-bit connections to left and right neighbors for systolic data and a 72-bit connection to the shared crossbar. Other connections not shown include four LED lines, two handshake lines connected to X0, two handshake lines connected to IF, and two handshake lines connected to the neighboring processing element. Other miscellaneous control pins such as reset, global tristate, clock, configuration, and readback are omitted.

The location of the memories and the major bus connections are designed to permit the PE to be divided into four "SPLASH 2-like" single-memory systolic virtual processing elements to improve pipelining and floor planning. With respect to pad locations, the memories are arranged along the top of the PE, the crossbar connection is centered on the bottom, and the left and right ring connections are on the left and right sides respectively. Both X1 and X2 processing elements are identical so that the same SIMD or systolic configuration can be easily replicated without redundant synthesis.

2.3.1.3 Control Element

The SLAAC control element, X0, consists of one Xilinx XC4085XLA-09 and two 256Kx18 bit synchronous SRAMs. The Xilinx 4085 contains a 56x56 array of CLBs for a 55K to 180K equivalent gates at clock rates up to 100MHz. X0 has two 72-bit ring connections, a 72-bit shared crossbar connection, and two 72-bit FIFO connections to the interface FPGA. Unlike SPLASH 2 and WildForce, X0 in the SLAAC architecture is designed to sit at both ends of the systolic array. X0 acts as the data stream manager for the architecture. Its primary mission is to read/write data from the FIFO module blocks implemented in the IF chip and pass this data on to the processing elements. The location of the memories and major connections in X0 are designed to allow the device to be split into a pre-processing section on the left, and a post processing section on the right half of the FPGA. If the memories are not used for pre- or post-processing, X0 can support two additional "SPLASH 2-like" virtual processing elements for a total of ten virtual processing elements on SLAAC-1.

2.3.1.4 Interface

The SLAAC-1 interface includes a Xilinx XC4062XLA-09 and several supporting components for clock generation and distribution, configuration, power management, external memory access, and system bus interfacing.

2.3.1.5 Clock

The SLAAC interface includes a clock generator tunable from 391 kHz to 100 MHz in increments less than 1 MHz. Clock distribution is separated into two domains. A processor clock (PCLK) drives the logic in X0, X1, and X2. PCLK is looped through the interface FPGA to support flexible countdown timers and single-step clocking. A memory clock (MCLK) drives the user memories and allows the host to access the memories while the PCLK is halted.

2.3.1.6 External Memory Bus

All of the user programmable memories in the SLAAC architecture are accessible from the host processor through an external memory bus. This feature guarantees a stable path to the memories for initialization, debugging, and retrieving results without depending upon the state of the user FPGAs. For each memory, a pair of transceivers isolates the address/control and data lines from the shared external memory bus. The transceivers are controlled from the IF chip.

For the SLAAC-1 architecture, the hardware team chose to implement a preemptive memory access strategy similar to that of SPLASH 2. In a preemptive memory access, the host interrupts the user FPGAs to read or write the memory. First, the global tristate (GTS) is asserted on the user FPGAs to prevent them from accessing the memories. Then, the PCLK is halted to stop the user logic. After the IF chip completes the memory transactions on behalf of the host processor, this process is reversed. The user FPGAs are unaware that the access has occurred. This greatly simplifies the user's design because exclusive access to the memory is assumed. No special states are required in user state machines for initialization or debugging.

Although a preemptive memory access strategy was chosen for SLAAC-1 in the interface implementation, the fact that the interface is within an FPGA allows exploration of other approaches. In addition to this, the fact that the SLAAC-1 memories and transceivers that implement the external memory bus are located on replaceable memory modules (see Figure 2-5) presents ample opportunity to experiment with alternate memory designs. Each memory module has 160 pins connected to one of the processing elements and a 40-pin connection to the external memory bus.

2.3.1.7 Configuration

The IF device is programmed on power-up by an EEPROM to provide a stable interface to the host. The EEPROM program pins are accessible to the host through a control/status register in IF. This enables in-system updates of the interface through software. The user programmable FPGAs in the system are configured from IF. X0, X1, and X2 can be programmed individually or in parallel. A simple slave bus configuration through a set of control/status registers is currently supported for SLAAC-1. However, there are two additional memories on the external memory bus dedicated as configuration and readback cache. The host can quickly load the configuration cache and the configuration can occur autonomously in IF, thus freeing up the host more quickly. An added benefit of placing the configuration memories on the external memory bus is that any or all of the ten user memories can be conscripted as configuration caches. Up to six complete SLAAC-1 configurations (including X0, X1, and X2) can be stored simultaneously on SLAAC-1 and selected with minimal effort from the host.

2.3.1.8 Readback

An integral part of rapid prototyping on reconfigurable architectures is the ability to debug a design on the hardware. The Xilinx readback facility is essential. The IF chip provides readback access to X0, X1, and X2 through a set of control/status registers. The user generates a readback trigger signal and readback data is stored in the configuration cache memory. Once readback is completed, the host can access the readback data from the configuration memory with direct memory reads.

2.3.1.9 FIFOs

Instead of dedicated hardware FIFOs, the SLAAC team decided to implement FIFOs within the IF chip. SLAAC-1 supports four input FIFOs and four output FIFOs inside the IF chip. The input FIFO port on X0 has 64 data, 4 tag, one empty flag, one read-enable, and two FIFO-select pins. The output FIFO port has the same pins with the exception of a full flag instead of an empty flag and a write-enable instead of read-enable. Selecting source and destination FIFOs is essential for the network-centric SLAAC system level architecture. The numbered FIFOs act as separate endpoints for ACS system channels. The user FPGA logic can simultaneously process a number of input and output streams and dynamically route data across multiple network channels on a cycle-by-cycle basis.

2.3.1.10 Power Management

Power consumption by FPGAs is a function dominated by clock rate and bit toggle rate. Since the user logic in X0, X1, and X2 has the potential of drawing too much power from the PCI slot and this behavior is application dependent, the SLAAC interface includes a power monitoring circuit. Power monitoring is accomplished using a current to voltage monitoring circuit on the +5V, +3.3V and +2.5V supply lines. Each circuit uses a LMC6482 operational amplifier and a low value current sensing resistor. Feedback resistors set the appropriate gain. These analog voltage levels are then monitored by a PICmicro™ 16715E microcontroller that has four A/D input channels available. Once a threshold level has been triggered the microcontroller interrupts the IF device. The IF design is able to halt the processor clock to stop the user FPGAs and interrupt the host.

2.3.1.11 Physical Implementation

SLAAC-1 is a full-sized PCI board designed for use in the RRP workstations. Although the initial release of the interface FPGA contains a Xilinx 33MHz 32-bit PCI core, the hardware is capable of supporting 64-bit PCI. Figure 2-4 contains a photograph of SLAAC-1 assembled in March 1999. In order from left to right, the large BGA devices are IF, X0, X2, and X1. The double-row of 100-pin connectors above and below X0, X1, and X2 support memory daughter card modules. A memory module is shown in Figure 2-5. Each memory module has four 256Kx18 synchronous SRAMs and the transceivers for the external memory bus. The memory module for the IF and X0 devices share one memory card since there are two memories for X0 and two configuration cache memories for IF.

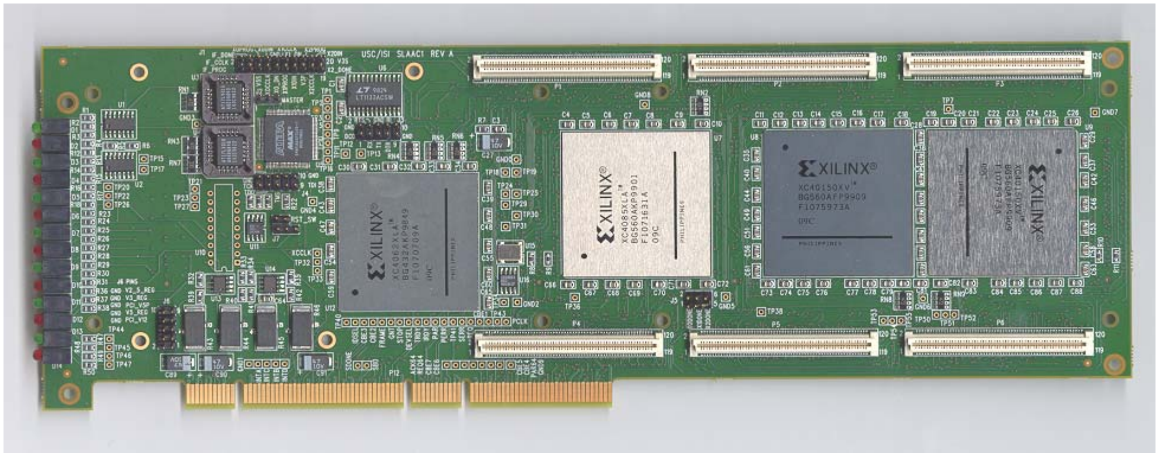


Figure 2-4 SLAAC-1 Photo

On the back of the SLAAC-1 board (not shown) are four systolic connectors for the high-speed data path through the X1 and X2 chips. The 64 data bits of the X0 to X1 and the X2 to X0 ring paths are shared with the systolic connectors. Additional pins from the X1 and X2 chips provide control for the respective external data sources.

Figure 2-5. SLAAC-1 Memory Module

2.3.2 SLAAC-2 (VME)

SLAAC-2 is a 6U VME mezzanine board designed to plug into a CSPI M2621S baseboard carrier. As shown in the SLAAC-2 architecture diagram in Figure 2-6, there are actually two SLAAC-1 compatible accelerators on the SLAAC-2 board, node A and B, each controlled by one of the two PowerPCs on the M2621S. A few modifications were necessary to the basic SLAAC-1 design to accommodate having two accelerators in an area not much larger than a single full-sized PCI board. However, most of these changes are not directly visible to the SLAAC-2 application designer.

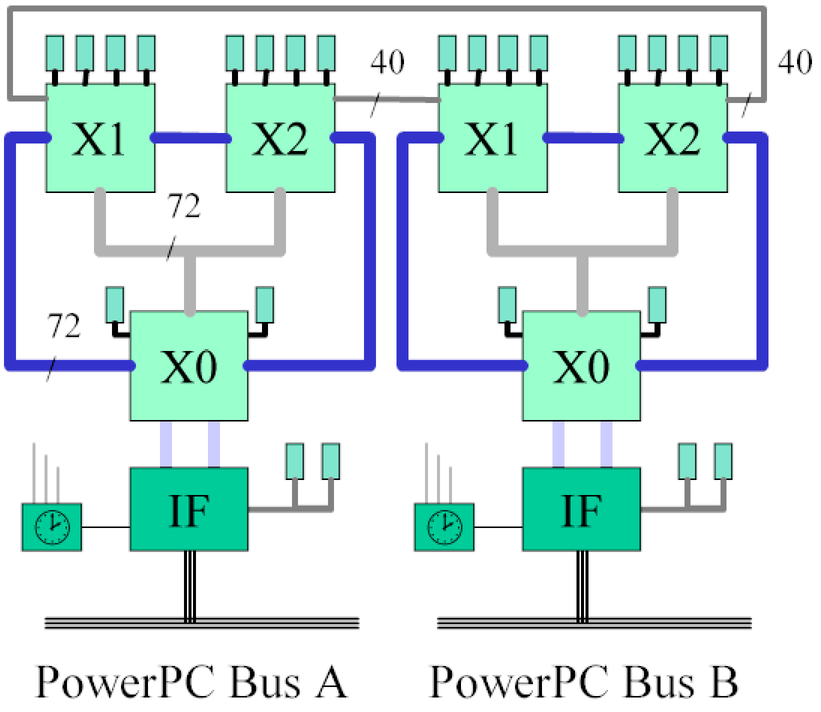


Figure 2-6. SLAAC-2 Architecture Diagram

One change to the SLAAC-2 design was that the Xilinx 4062 IF device on SLAAC-1 was replaced with Xilinx 4085s. The extra I/O pins available on the 4085 were needed to accommodate the non-multiplexed 64-bit PowerPC bus. Other modifications were made to save space on the board, including combining the power management, IF boot EEPROMS, and the reference oscillator. The external memory bus was the only casualty to compute density visible to the user. There was insufficient area available for the transceivers necessary to isolate the external memory bus during normal user FPGA operation. It was decided that since the SLAAC-1 and SLAAC-2 user FPGAs are bitfile compatible, debugging the memories could happen on a PCI board and was not essential in the VME platform. The only consideration for the application designer is that the memories will have to be loaded from within X0, X1, and X2.

Also shown in the SLAAC-2 architecture diagram are two 40-pin busses between X1A and X2B, and X1B and X2A. The spare pins used for controlling the external systolic connectors in SLAAC-1 were used in SLAAC-2 to bridge the two "independent" designs. Although the A and B designs have separate tunable clock synthesizers, a side-effect of having a single reference oscillator on SLAAC-2 will allow the two designs to operate synchronously with each other. In any event, spare pins in the compute FPGAs are used so that X1A and X2A have access to the B design's clock and vice versa with X2A and X2B. This permits cooperation between the two adjacent nodes. Figure 2-7 is a photo of the component side of SLAAC-2. A total of six FPGAs is visible on this side. Two additional FPGAs on the back are not shown. The long connectors visible that nearly span the length of the board are the PowerPC bus connectors.

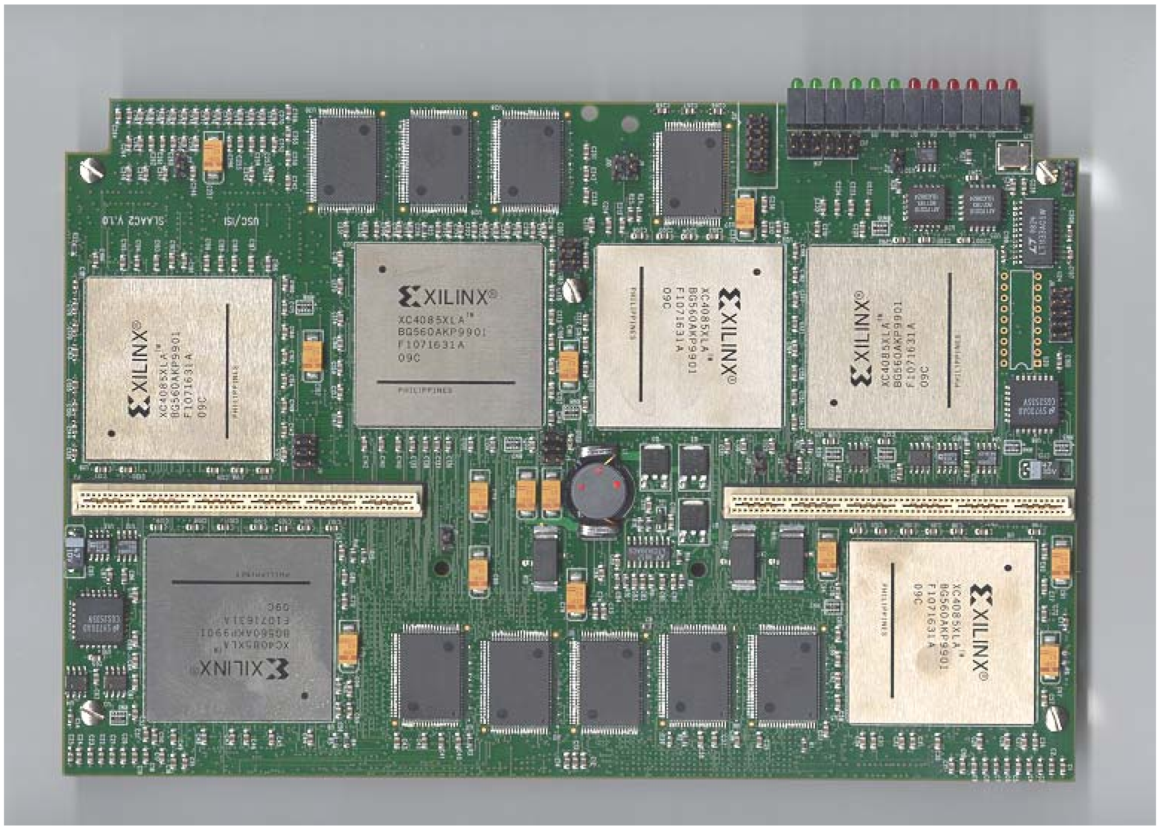


Figure 2-7. SLAAC-2 Photo

2.3.3 <u>SLAAC-1V (PCI)</u>

SLAAC-1V was developed in 2000 to explore the capabilities of new Xilinx Virtex devices. It was designed to be source-code compatible with SLAAC-1 and SLAAC-2 at the VHDL level.

2.3.3.1 Processing Elements

The SLAAC-1V architecture is shown in Figure 2-8. The three Virtex 1000 FPGAs (denoted as X0, X1, and X2) are the primary processing elements for a total of three million gates. Each Virtex has 12,288 basic logic cells and 32 internal 4-kbit blocks of dual-ported SRAM. Similar to earlier SLAAC boards, the processing elements are connected with a 72-bit "ring" and a 72-bit shared "crossbar" bus. The width of both buses supports an 8-bit control tag associated with each 64-bit data word. The direction of each line of both buses can be controlled independently.

2.3.3.2 Control Element

SLAAC-1V doesn't have a separate PCI interface chip. Instead it has a separate Virtex 100 configuration controller (CC) with configuration SRAM and FLASH. When the board is powered up, the configuration controller loads the default X0 configuration from FLASH. The host then writes to the SRAM and asks the CC to reprogram all or part of X0. SLAAC-1V supports partial reconfiguration, in which part of an FPGA is reconfigured while the rest of the FPGA remains active and continues to compute.

About 20% of the resources in the X0 FPGA are devoted to the PCI interface and board control module. The remaining logic can be used by the application. The default X0 module uses the Xilinx 32-bit 33MHz PCI core. It provides high-speed DMA (Direct Memory Access), data buffering, clock control (including single-stepping and frequency synthesis from 1 to 200 MHz), user-programmable interrupts, etc. The default module has achieved DMA transfer rates of over 1 Gbit/s (125 MB/s) from the host memory, very near the PCI theoretical maximum. The bandwidth for SLAAC-1V using the 64-bit 66MHz PCI controller (using the Xilinx 64-bit 66MHz core) has been measured at 2.2 Gbit/s.

The user's design located in X0 is connected to the PCI core via two 256-deep 64-bit wide FIFOs. The DMA controller located in the interface part of X0 can transfer data to or from these FIFOs as well as to provide fast communication between the host and the board SRAMs. The DMA controller load balances input and output FIFOs and can process large memory buffers without host processor interaction. Current interface development includes managing memory buffer rings on the FPGA to minimize host interrupts on small buffers.

2.3.3.3 Memories

The processing elements are connected to ten 256K x 36-bit SRAMs. X1 and X2 are each connected to four SRAMs, while X0 is connected to only two to accommodate the PCI interface. The memory cards have passive bus exchange switches that allow the host to directly access all memories through X0.

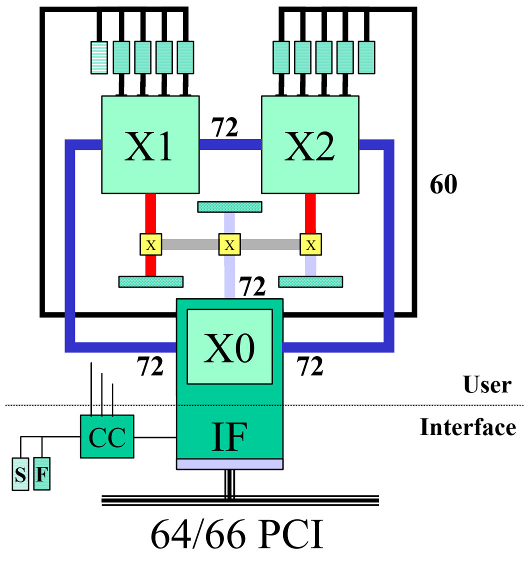


Figure 2-8 SLAAC-1V Architecture

2.3.3.4 External I/O

SLAAC-1V also added bus exchange switches to the "crossbar" port. Each chip can switch between the shared bus and an external I/O connector. X0 has a connector on the front. X1 and X2 have connectors on the back.

2.3.3.5 Physical Implementation

SLAAC-1V is a full-sized 64-bit 66MHz PCI card. The memories and memory switches are located on two memory daughter cards (not shown). The vertically mounted connector on the left side is the X0 external connector. It is closest to the back of the PC.

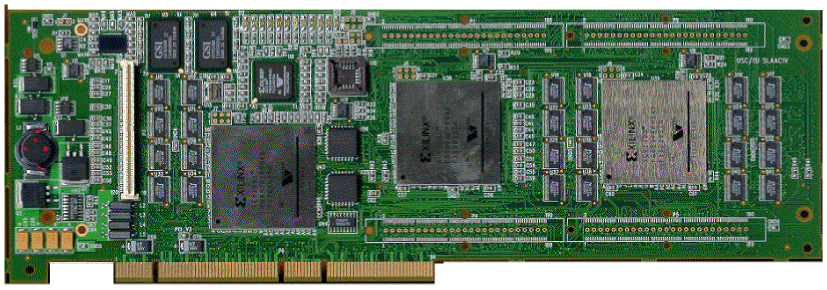


Figure 2-9 SLAAC-1V Photo

2.3.4 <u>Osiris Virtex-II PCI Accelerator</u>

Osiris was developed in 2001 to exploit the computational density of Xilinx Virtex-II. It includes all of the optimizations on our wish list from our experience with SLAAC-1V. The biggest requests were for more onboard memory (on the order of one Gigabyte) and maximum SRAM performance. Approximately 30 Osiris boards were produced. Osiris was licensed to Atlantic Coast Telesys and they are now being produced commercially.

2.3.4.1 Processing Element and Memories

Instead of three chips, this full-sized PCI card had one large Virtex-II 6000 FPGA for approximately 6M logic gates. That number is a bit misleading since the hard multipliers and dedicated block ram are counted as gates. The board had considerably more onboard memory, ten banks of 512K x 36-bit 250MHz SRAM and two SODIMM sockets with up to 512MB of PC133 SDRAM. We encountered a performance hit for having bus exchange switches and an external connector, so for Osiris we tried to optimize SRAM placement. Instead of external switches, we developed a programmable drop-in ring inside XP.

2.3.4.2 Control Element

In some ways, this board more closely resembles SLAAC-1 in that we separated out the interface logic to a Virtex-II 1000. It supports PCI 32/33 (3V only), PCI 64/66, or PCI-X. The advantage of this is that it doesn't require a user to compile in a PCI core with the flakiness of place and route tools, which sometimes caused problems in SLAAC-1V. Osiris added a standard PCI Mezzanine Card (PMC) connector for standard external I/O cards or processor cards. The interface chip actually bridges between two PCI busses; the left-hand interface connects to the host and the right-hand interface connects to PMC. This provides a unique capability to allow a host to control the PMC card directly with standard device drivers, and let the PMC card DMA data directly to the compute element without interfering on the host PCI bus. This approach also allows the XP user design to be ignorant of data origination. XP interfaces to a DMA FIFO only.

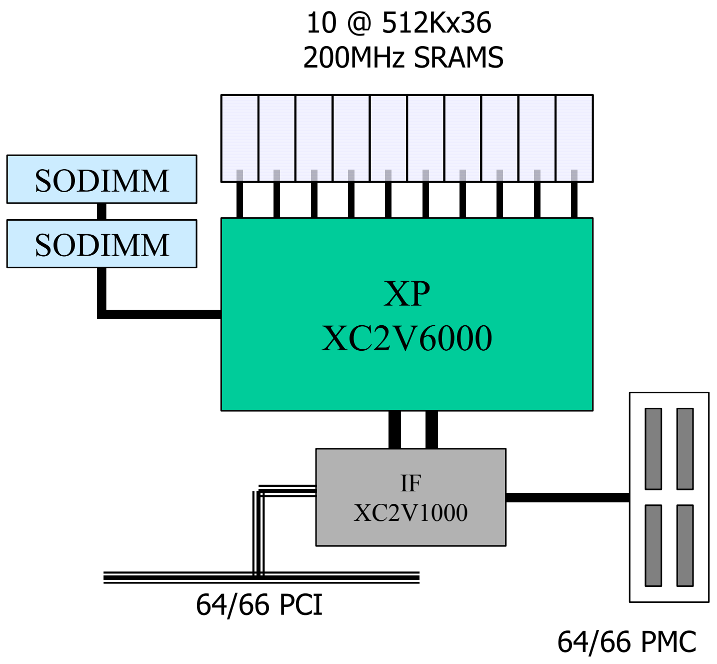


Figure 2-10 Osiris Architecture

2.3.4.3 Physical Implementation

An Osiris photo is shown below. The large chip in the center of the board is the Virtex-II 6000 surrounded by 10 SRAM banks. Two SODIMM sockets (one on back) are to the right of the compute element. Centered over the PCI connector is the interface chip. Between IF and XP are the PMC connectors. All components to the left of the PMC connector follow the PMC no-fly-zone specification.

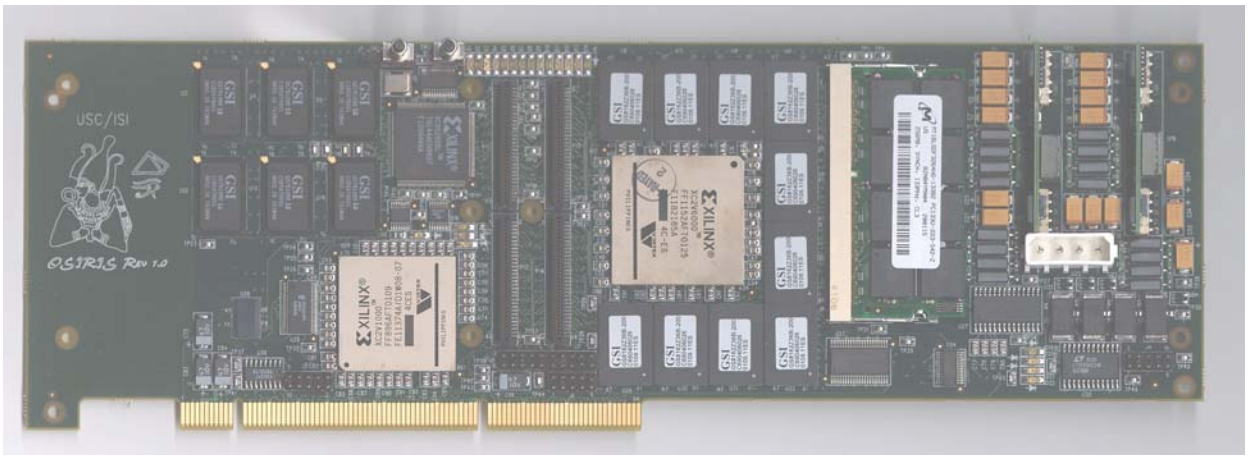


Figure 2-11 Osiris Photo

2.3.5 <u>PipeRench PMC Board</u>

Part of the SLAAC project charter was to assist other DARPA ACS projects demonstrate their technology. We produced a PMC daughter card for Carnegie Mellon University. It had a Virtex-II 1000, two banks of SRAM, and two PipeRench sockets. Ten PipeRench PMC boards were produced and delivered to CMU for their demonstrations.

2.4 Milestone Summary

2.4.1 <u>1998</u>

- In early 1998, we developed the RRP and DRP concepts and did some early implementation studies targeting a Myrinet VME baseboard. When it became clear that Myricom, Inc. was having trouble building it (under another DARPA contract) we looked for COTS alternatives.
- In the middle of 1998, we started architecting a PCI and VME board in parallel (SLAAC-1 and SLAAC-2). We negotiated with CSPI, Inc. to fabricate a VME carrier board for SLAAC-2 based on their commercial M2641 Quad PowerPC board.
- VHDL simulation models of the boards were delivered to SLAAC application teams.
- The SLAAC-1 prototype was produced in December 1998.

2.4.2 1999

- In early 1999, the SLAAC-2 prototype was completed. Hardware testing and firmware development dominated ISI effort throughout the first half of 1999.
- ISI delivered SLAAC-1 PCI boards and NT software to BYU, UCLA, Virginia Tech, LMGES, and Los Alamos in October.
- An additional ten SLAAC-1 PCI (Rev B) boards were fabricated. Several of these boards were delivered to external organizations.
- ISI had three prototype SLAAC-2 VME boards operational at ISI, but they were pretty temperamental in different VME chassis. ISI revised SLAAC-2 and replaced the culprit component. ISI fabricated ten SLAAC-2 VME (Rev A) boards and assembled one with Xilinx 40150s for testing.
- ISI completed the architectural design for the SLAAC-1V Virtex PCI board and assembled three prototypes in December 1999.

2.4.3 2000

- ISI focused on SLAAC-1V hardware production and firmware development and testing in 2000.
- ISI added Linux and Solaris driver support to SLAAC-1 and SLAAC-1V.
- ISI delivered a large number of SLAAC-1V boards in mid-2000 to internal and external customers.
- ISI delivered a SLAAC-2 to Integrated Sensors Inc. for HRTExpress tool integration under another DARPA effort.
- ISI broke the 2Gb/s bandwidth barrier with SLAAC-1V using 64-bit 33MHz PCI.

2.4.4 2001

- In early 2001, ISI began development on Osiris, a Virtex-II 6000 board. This was the third and final generation of SLAAC technology.
- SLAAC-1V successfully ran 64-bit 66MHz PCI interface.
- ISI fabricated ten prototype Osiris with engineering sample Virtex II chips.
- ISI began development of a PipeRench PMC daughter card for CMU.

2.4.5 2002

- ISI produced 20 Osiris boards and distributed them to the community. ISI completed 64/66 PCI firmware, device driver, simulation models, etc.
- ISI licensed Osiris technology to Atlantic Coast Telesys. They will be producing Osiris commercially in late 2002.
- ISI developed a PMC daughter card for PipeRench.

2.5 Results

2.5.1 SLAAC-1

SLAAC-1 had some interesting capabilities and defined the baseline architecture we would use in all boards that followed. It's primary purpose was a relatively low cost development platform for deployed SLAAC-2 systems.

Customer	**ACS Project**	**Delivered**
BYU	SLAAC	2
ISI	DEFACTO	1
ISI	SLAAC	3
LANL	SLAAC	1
NSA / Alan Hunsberger	N/A	2
UCLA	SLAAC	1
VT	SLAAC	1
TOTALS:		**11**

Table 2-1 SLAAC-1 Hardware Deliveries

Table 2-1 shows the SLAAC-1 requests and deliveries. Customers listed as N/A had purchased boards at-cost from ISI. No ACS funds were expended for production of those boards.

2.5.2 SLAAC-2

SLAAC-2 was our target deployable reference platform. It demonstrated that a VME slot could carry a very dense amount of FPGA logic and still be programmable. The feature of bitfile compatibility between PCI and VME boards was later replicated by Annapolis Microsystems on their commercial product line. It is somewhat surprising to the team that this board didn't transition into more SLAAC applications. However, natively interfacing to PowerPC bus and dealing with VxWorks as a development environment delayed customer use of this board until after interest had moved to SLAAC-1V Virtex. The following table shows the external deliveries of SLAAC-2.

Customer	**ACS Project**	**Delivered**
LMGES	SLAAC	1
Integrated Sensors	SLAAC	1
TOTALS:		**2**

Table 2-2 SLAAC-2 Hardware Deliveries

2.5.3 SLAAC-1V

The primary goal of SLAAC-1V was to leverage denser Virtex FPGA devices for our application demonstrations. This goal was achieved. A secondary research goal was to explore fast runtime reconfiguration. There has been significant algorithm work in past years for this design approach, such as evolvable hardware, without much support from commercial hardware vendors. SLAAC-1V was optimized for this sort of application. Most of the mainstream application demonstrations we delivered under SLAAC were static. However, SLAAC-1V fast RTR capabilities were recently found to be useful to LANL for a Single Event Upset simulator. Under DOE funds, LANL has been using SLAAC-1V reconfiguration to inject bit errors in the configuration stream and use fast readback to characterize failures. LANL invested internal funds to assemble the excess SLAAC-1V PCB with a socket for proton radiation testing this year. The following table shows the SLAAC-1V requests and deliveries. Customers listed as N/A have purchased boards at-cost from ISI. No ACS funds were expended for production of these boards.

Customer	**ACS Project**	**Delivered**
AFRL-WP	N/A	2
ARFL Rome / Ralph Kohler	N/A	2
BYU / Nelson, Hutchings	SLAAC	3
CMU / Herman Schmitt	PIPERENCH	1
FTL Systems	CAMERON	1
GMU / Tarek El-Ghazawi	LUCITE*	2
ISI	SLAAC	4
LANL / Maya Gokhale	SLAAC	3
LANL / Paul Graham*	SLAAC/DOE	6
MIT / Anant Agarwal	DIS/RAW	1
NSA / Alan Hunsberger**	N/A	2
NSA / Brian Weeks	N/A	1
Sandia	SLAAC	4
U Cinn / Ranga Vemuri	ARC	1
USC / Victor Prasanna	MAARCII	1
UTENN / Don Bouldin	CHAMPION	1
VT	N/A	1
VT	DRACS	1
VT	SLAAC	1
XILINX (VT)	LOKI	1
TOTALS:		**39**

Table 2-3 SLAAC-1V Deliveries

*LANL paid to produce 6 boards for Virtex radiation testing in 2002. The chips were donated by Xilinx and SLAAC provided bare PCB and minor excess components.

**NSA's LUCITE initiative has funded Drs. Tarek El-Ghazawi (George Mason University) and Nikitas Alexandridis (George Washington University) to investigate distributed resource management tools for adaptive computing systems in high-performance computing environments. In a discussion with Alan Hunsberger from NSA, Dr. Phillips approved the transfer of two SLAAC-1V boards to George Mason University for this effort.

2.5.4 Osiris

Osiris was developed to leverage Virtex-II device advances and also meet the need for onboard memory with two SODIMM sockets. This board was really optimized for performance from everything we had learned from earlier boards. In some ways, this design more resembles SLAAC-1. We separated out the host and external I/O interface firmware to a Virtex 1000 to simplify the programming environment for application designs. The idea is that data could stream from a Myrinet PMC card, a SCSI card, or the host without the "compute engine" FPGA to be aware of it. We added a standard PMC connector to make off-the-shelf I/O cards a possibility for this board and having a left-hand and right-hand PCI bus with a bridge means that the host workstation can handle all of the mucky device driver interfaces. This was a limitation of earlier boards such as Pammette because the FPGA had to implement an entire device driver as if it were a processor. This step isn't necessary on Osiris.

Below is a table of the production Osiris deliveries. The ten engineering sample boards are also functional and have been loaned out to supplement application development.

Customer	**ACS Project**	**Delivered**
BYU / Nelson, Hutchings	SLAAC	2
CMU / Herman Schmitt	PIPERENCH	2
ISI / DARPA PCA AMP funded	N/A	2
ISI	SLAAC	3
LANL / Maya Gokhale	SLAAC	2
NSA / Alan Hunsberger	SLAAC / N/A	2
NUWC / Bob Bernecky	SLAAC	6
VT	SLAAC	1
TOTALS:		**20**

Table 2-4 Osiris Deliveries

The Osiris platform design and associated software has been licensed by and will be sold commercially by Atlantic Coast Telesys. A product brochure is included in the appendix.

2.6 Related Publications

- Tim Grembowski and Roar Lien and Kris Gaj (HP) and Nghi Nguyen and Peter Bellows and Jaroslav Flidr and Tom Lehman and Brian Schott (HP), *Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512,* Lecture Notes in Computer Science, Vol. 2433, p. 75-??, 2002.
- Stephen P. Crago and Brian Schott and Robert Parker (HP), *SLAAC: A Distributed Architecture for Adaptive Computing*, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 286-287, IEEE Computer Society Press, April 1998.
- Brian Schott and Chen Chen and Steve Crago and Joe Czarnaski and Matt French and Ivan Hom and Tam Tho and Terri Valenti, *Architectures for System-Level Applications of Adaptive Computing,* IEEE Symposium on FPGAs for Custom Computing Machines, pp. 270-271, IEEE Computer Society Press, April 1999.

- Pawel Chodowiec, Kris Gaj, Peter Bellows, Brian Schott: *Experimental Testing of the Gigabit IPSec-Compliant Implementations of Rijndael and Triple DES Using SLAAC-1V FPGA Accelerator Board.* ISC 2001: 220-234
- Mark Jones, Lucas Scharf, Jon Scott, Christian Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas, and Brian Schott, "Implementing an API for Distributed Adaptive Computing Systems," Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, April, 1999.

3 FINAL STATUS REPORT ON THE ACS RUNTIME SYSTEM

Dr. Peter Athanas, Dr. Mark Jones
Virginia Polytechnic Institute and State University

3.1 Introduction

Developing an application on an adaptive computing board requires two primary tasks: (1) developing the hardware design for the adaptive computing device and (2) developing the high-level language program for the host computer that controls the adaptive computing device. Each adaptive computing device has a different interface, requiring a different program to be written for every device. Developing an application on an adaptive computing system consisting of multiple boards requires not only the hardware design and interface program, but also the networking code to move data and control information between the boards over a high-speed network. Not only is such code dependent on the nature of the network, it also requires a very different skill set for the designer.

Given this situation, there was a strong need for a solution that would provide a single programming target for the interface to the entire range of adaptive computing boards as well as provide an easy-to-use interface for a multi-board adaptive computing system. Balanced against ease-of-use is the need for high performance; adaptive computing system users will not adopt any solution that comes at a large sacrifice in performance.

3.2 Technical Approach

To meet these needs, ISI and Virginia Tech developed a specification for the ACS API, an application-programming interface for systems of adaptive computing devices. The ACS API provides a consistent interface for the full range of adaptive computing devices. In addition, the ACS API provides an interface for controlling the configuration and communication of a multi-board system. Most importantly, this API is designed to enable high-performance implementation.

The following briefly outlines the nature of the API; the complete API specification is documented in the Appendix. The ACS API views each adaptive computing board as a *node* and each logical link between boards as a *channel*. The API provides a set of subroutines that allows a user to specify a *system* consisting of the set of nodes to be used in a computation as well as the channels those nodes will use for communication. In addition, an XML-based tool is available to allow users to easily construct a system without any programming required (documentation for this tool is provided in the Appendix).

After specifying a system, the host program can use the API to specify the appropriate configuration for each node in the system (in the case of FPGAs this consists of specifying the configuration bitstream files). The API implementation then handles transporting the configurations to the correct nodes. In support of high-performance, the API provides support for run-time reconfiguration (RTR) of the nodes. One of the primary overheads associated with RTR is the cost of getting the new configuration to the node. In support of this, the API provides for caching configurations near the board, rather than resending them over the network or over a PCI bus. The API also provides support for both host-driven and data-driven RTR.

Once the system is specified and configured, the API provides for routines to move data to nodes via either direct memory reads/writes or via the FIFO-oriented channels. These routines are specified such that they are non-blocking and can be easily grouped to allow for high-performance implementation. In addition, there are miscellaneous functions that handle interrupts, read/write registers, and control the clock frequencies on nodes.

As part of the SLAAC team, Virginia Tech designed and developed a high-performance implementation of the ACS API. To achieve maximum portability, this design is implemented as three separate layers. The system layer provides the API interface to the user, handling all direct interactions with the user. It is the portion of the system least likely to change when migrated to new operating systems, boards, or networks; it currently runs under Linux and Windows NT. The control layer is a compact program that runs on each node in the system, acting as the control for all networking in the system; it acts as a bridge between the system layer and the yet-to-be-described node layer. The control layer's implementation may be modified for a different networking environment. Finally, the node layer provides the direct interface to an adaptive computing board. The node layer is completely dependent on the interface to the board; a new node layer is written for each type of adaptive computing board. A complete specification of the node layer is included in the Appendix. Only the node layer must be modified to port the system to a new board.

To be judged successful, the ACS API implementation must provide interfaces to several types of adaptive computing boards, allow for heterogeneous systems of boards to be easily specified, and not require substantial runtime overhead. As noted in the results section, these goals have been achieved.

3.3 Milestone Summary

- November 1998: Work on specifying the ACS API begins.
- April 1999: An initial version of the ACS API implementation is complete. The API runs on a cluster of Annapolis Micro Systems WildForce boards connected by Myrinet. A simple debugging system is available for single boards through the ACS API interface.
- July 1999: Test programs for the ACS API are implemented, including programs to measure all aspects of the current ACS API implementation as well as their performance. These programs are distributed with the API.
- October 1999: The API is ported to Linux and support added for the AMS WildOne, SLAAC1, and Sanders RCM board.
- April 2000: The API specification is extended to support caching of configurations. The ACS API implementation is demonstrated on a heterogeneous multi-board system.
- July 2000: Extended documentation on the API use and implementation is provided. The ACS API implementation is extended to the AMS WildStar. Initial support for the SLAAC1-V is provided.
- June 2001: A new version of the API specification and implementation is released with support for caching along with bug fixes.

- October 2001: Support for the commercial Celoxica board is included in the implementation. In addition, initial RTR experiments on the SLAAC1-V are performed in the API.
- December 2001: Initial version of the ACS API debugger is released. Reconfiguration from onboard memory on SLAAC1-V via ACS API accomplished.
- April 2002: Support for Osiris board included in ACS API.
- June 2002: The API is demonstrated on a heterogeneous cluster of Osiris and SLAAC1-V boards. Final versions of the API and debugger are released.

3.4 Results

The specification and implementation of the ACS API, including source code, are available at www.ccm.ece.vt.edu. The API has been ported to the following boards: AMS WildForce, AMS WildStar, AMS WildOne, SLAAC1, SLAAC1-V, Osiris, Sanders RCM, and Celoxica. In addition, a graphical and textual debugger based on the ACS API is also available; it works on any system to which the ACS API has been ported. All of these tools and source code are available to the community free of charge and without restriction.

The implementation has been demonstrated on two heterogeneous systems, in addition to homogeneous clusters of boards. A multi-board filter was implemented across an AMS WildForce, a SLAAC1 board, and the Sanders RCM board using the ACS API. Later, a multi-board pattern matching system was implemented on a system with an Osiris board and multiple SLAAC1-V boards. Note that in both cases, the configuration of the boards could easily be changed through the API.

The ACS API implementation has been evaluated in terms of the runtime overhead. A single board FIFO-based, implementation of the pattern matching application running on the Osiris board had no perceptible overhead when using the ACS API versus the native API. The same application running on the SLAAC1-V had an overhead of less than 50%, all attributable to a temporary issue in the device driver (to be remedied). A memory-based implementation of the same application had an overhead of approximately 30% for each of the boards. When running on a remote Osiris board, the system achieved a throughput of over 90% of the network capacity (where the network is the bottleneck in the system).

3.5 Related Publications

- Mark Jones, Lucas Scharf, Jon Scott, Christian Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas, and Brian Schott, "Implementing an API for Distributed Adaptive Computing Systems," Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, April, 1999.
- Kuan Yao, "Implementing an Application Programming Interface for Distributed Adaptive Computing Systems," M.S. thesis, Virginia Tech, 2000.
- Zahi Nakad, "High Performance Applications on Reconfigurable Clusters," M.S. thesis, Virginia Tech, 2000.
- William Worek, "Matching Genetic Sequences in Distributed Adaptive Computing Systems," M.S. thesis, Virginia Tech, 2002.

4 FINAL STATUS REPORT ON THE SAR/ATR CHALLENGE

Dr. Brian Bray
Sandia National Laboratories

4.1 Introduction

SAR ATR is a computationally challenging problem, even with restricted operating parameters and hierarchical processing. The surveillance challenge of SAR ATR was to support a 1 ft. resolution SAR sensor with a coverage rate of 40,000 sq. nm. per day. Before the start of the program in 1995, DARPA was funding the development of the Global Hawk UAV. The SAR sensor was to have a wide area coverage of 40,000 sq. NM. per day but with only a 1 meter resolution. We postulated that a 1 ft. resolution mode would eventually be used for the wide area search since around 1 ft. resolution is need to identify all but the largest military vehicles. To keep the 40,000 sq. NM. per day area coverage, a 1 ft. resolution SAR sensor would be generating a data stream of about 40 Mpixel/sec, given expected imaging parameters such as oversampling and depression. A 40 Mpixel/sec input stream would be quite an increase of processing requirements over the best current system of the time which was 1 Mpixel/sec system developed by Sandia National Laboratories. Besides having to process 40 times more data than any existing system, a system or part of the system might have to be small enough to fit in the UAV. A 40 Mpixel/sec SAR image feed would actually require a 10,240 Mbit per second data link to send the imagery to the ground stations, since each pixel is a complex pair of 16 bit I and 16 bit Q values. This bandwidth requirement was not deemed feasible for a UAV air-to-ground wireless link. Hence, some portion of the ATR processing chain would need to be onboard the UAV to extract regions of interest for high fidelity lossless transmission, while most of the SAR imagery is compressed heavily using intelligent bandwidth compression. The onboard ATR processor would be volume and power constrained.

Constructing a system with forty times the performance of the best existing system which was comprised of several VME chassis, is not too bad since about 6 generations of semiconductor technology improvements would eventually make that small enough. Unfortunately, the number of targets that are being searched for needs to be increased from the then current 6 to 30 or so. This would basically increase the computational load on the system by a factor of 5. Now the system is about 200 (40 x 5) times existing technology.

Constructing a system two hundred times the performance of the best existing system, would now take about 8 generations of semiconductor technology improvements. Adding to the computational demands of the system is the growing computational complexity of the algorithms. Since Sandia had success in the finding targets out in the open, algorithm development was in the process of being switched to become more robust to target variations, camouflage, and concealment and deception scenarios. These initial algorithms were approximately one hundred times more computationally intensive than their less robust counterparts. Now the overall system needed to support the surveillance challenge of covering 40,000 sq. NM. with a 1 ft. resolution SAR sensor would be 20,000 times more computationally demanding.

At the time Sandia National Laboratories was working with Northrop Grumman to integrate the Sandia National Laboratories SAR ATR with new advanced radar modes on the Joint STARS test aircraft with a goal of upgrading the production fleet at a later time. The SAR sensor would not have as high a Mpixel/sec rate as the Global Hawk. For Joint STARS the SAR sensor mode is a secondary sensor mode that is cued by the primary sensor mode so not as high a Mpixel/sec rate is needed for the Joint STARS con-ops and mission. Hence only the increase in targets and more robust algorithms would be needed for supporting this task. The increase in target requirements by five and the increase in robustness (100 times computational increase) would result in an overall system increase of 500 times.

Hence a goal of 500 times improvement would be directly applicable to the SAR ATR mission for Joint STARS and a longer term goal of supporting a potential enhancement of the wide area search capability of the Global Hawk UAV.

Under the DARPA HPC program, Sandia National Laboratories had migrated from custom systolic processing modules to an all COTS heterogeneous multicomputer system for the SAR ATR processing. Custom systolic hardware was used because it was the fastest way to do the computations. However, the custom hardware was not getting faster since there was no money to re-spin the ASICs to the latest generation of silicon technology. Even if there were money to re-spin the ASICs to newer silicon, eventually the systolic design would be limited by off-chip and board-to-board synchronous clock rates. The ASICs did brute force bit and byte level correlation and the boards had smart flexible backends that were implemented using Xilinx FPGAs. Additionally, the newer more robust algorithms under-development did not use full kernel correlations. Preliminary analysis showed that FPGAs could achieve significant performance gains over that of CPUs given enough computational resources and memory bandwidth.

There are several things that enhance or detract from the performance of ACS over CPUs. The first thing to attack with ACS to get performance was to exploit spatial parallelism in the algorithms. In this application domain, most algorithm kernels have greater than 100X spatial parallelism because the same computation is repeated for each pixel in the image. In CPUs, the spatial parallelism is limited to one since there is only one thread of computation, while the spatial parallelism in ACS that is exploitable is limited by: (1) on-chip logic and routing resources; (2) on-chip memory size and bandwidth; and (3) external memory size and bandwidth. The second thing to attack with ACS is temporal parallelism which is the explicit execution of the equation versus the emulation that a CPU does by executing extra instructions to fetch, store and control the computation (e.g. loads, store, branches). A detraction of ACS performance is the difference in clock rate between CPUs and ACS (e.g. speed of highly tuned critical paths hardwired in silicon versus signal paths that are going over connections that are programmable). Another detraction for ACS performance is the need for a host to control the ACS board. We attacked this problem by trying to overlap communication with computation. A final detraction of ACS, but not a performance issues, is that the design effort is significantly larger and more drawn out. We attacked this problem by using graduate students and new design techniques that their professors were developing.

Our demonstration plan scaled down the surveillance challenge to minimize the initial hardware investment, yet was large enough to demonstrate a significant capability which would exist at the end of the program in a small volume (1 cu. ft.). The first demonstration was to occur in 1998 and use first generation embedded ACS to accelerate a system using baseline algorithms. In

1999, the second generation embedded ACS was to be used to accelerate some of the more robust ATR algorithms under development for the Joint STARS program. In 2001, the third generation embedded ACS hardware would accelerate the final set of Joint STARS algorithms. The volume-performance improvement goal was 500X over the 1996 baseline using more robust 2001 algorithms with an increased number of target configurations for both systems.

4.2 Technical Approach

In this section we will discuss the technical approach used to solve the problem, performance goals, metrics for success, the main components to be developed and components leveraged from previous work or other programs.

The SAR ATR surveillance challenge was going to heavily leverage the system infrastructure of the SAR ATR system that is being developed for Joint STARS and to piggyback on demonstrations if possible. The main hardware components to be developed were three generations of ACS boards leveraging off the commercial chip and experimental chip developments of other programs. Each generation of ACS boards was to have a development version for researchers and an embedded version for deployment.

One option for the first generation was to utilize medium size FPGAs with multiple memory ports and direct connection to a Myrinet LANai for local control and some processing. The second option was to utilize large existing style FPGAs like the Xilinx 4000 series with several ports to off-chip memory and use a host CPU to control most of the processing and networking. In the past researchers had typically hooked up only one memory port to each FPGA since the FPGAs were small, but FPGAs were going to be big enough that the computational capabilities would not be satisfied with the memory bandwidth provided by a single memory port.

The second generation was hypothesized to have even larger FPGAs like the Xilinx 4000 series but with the addition of blocks on on-chip RAM and multiplier support. The on-chip blocks of memory were deemed crucial to supplying memory bandwidth to the increasing computation capabilities of the FPGA. Analysis showed that SAR ATR correlations have most of the memory bandwidth requirements met with small blocks of memory. This memory bandwidth was especially important since now more spatial computational parallelism was going to be utilized since there would be significantly more on-chip computational resources.

The third generation was hypothesized to be several possible configurations since predicting technology that far out is somewhat uncertain. One of the potential third generation main components and most risky was to be FPGAs that reconfigured on a clock-by-clock basis. A second of the potential third generation main components was that a processor core with FPGA and memory resources tightly coupled. This way, FPGA like resources could be used to accelerate parts of the computational loops that made sense without occurring much overhead for interacting like with a custom accelerator or having to have the FPGA process the entire loop. The final and least risky of the potential third generation main components was that of the second generation FPGA that is enhanced further with complete microprocessor cores. This would reduce several bottlenecks in an attached accelerator model. First, much of the control of the FPGA processing could be done locally and autonomously using the local microprocessor. Second, low throughput tasks on some of the front-end or back-end processing that would required either large amount of FPGA design effort or more host CPU processing and interaction could be done on the local microprocessor core.

Sandia National Laboratories had primarily four main tasks: First, to describe algorithms and work with researches for mapping efficiently to the available generation of ACS hardware; second, to develop the software modifications to the SAR ATR system to utilize these accelerators; third, to carry out performance and correctness validations; and fourth to perform laboratory and if possible deployed demonstrations.

4.3 Milestone Summary

The milestone summaries are broken up by years. Within each year a team member and the algorithm part they were working on is listed followed by what was accomplished.

4.3.1 <u>1998</u>

- During this time, generation 1 FPGAs (e.g. Xilinx 4K family) were being used to accelerate the baseline algorithms. The algorithms were being mapped to a variety of FPGA boards since the SLAAC-1 board did not exist yet.
- Myricom & SLD: Sandia worked with Myricom to implement the SLD algorithm in an AT&T Orca FPGA on custom daughtercards that plugged into a custom Myricom switch board. Four nodes would fit in a single 6U VME slot. The nodes were be controlled by the LANai networking interface which also did some of the pre and post processing. These nodes used a byte wise computation model that was similar to how a CPUs would perform the SLD calculation. However, multiple processing locations were computed in parallel (spatial parallelism) by using a systolic streaming mode. The design ran at 40 MHz and had 15 processing elements in a single FPGA running in parallel. The results and design details are in the attached FCCM paper and the Myricom technical report. This board with two CSPI PowerPC 300MHz quad multicomputer boards were used to meet the 1998 performance target.
- UCLA & SLD: UCLA used a prototype board that was to mount on the Myricom LANai baseboard which was eventually abandoned. UCLA used a bit slice approximation algorithm with systolic streaming to perform the SLD calculation. The calculation was the same as how Sandia had designed the custom hardware in 1991. The unique thing was that UCLA was using significantly less hardware and performing multiple correlations at the same time by using shared adder trees that were unique to each template. Dynamic runtime reconfiguration would occur depending on what templates needed to be run. Even if FPGAs that reconfigure in one clock cycle were to become available, the performance was good but not nearly as good as the approach used by the Sandia and Myricom team for SLD. This negative result showed that the approach used by the Sandia and Myricom team was what should be used future ACS implementations for other correlation type algorithms that SLAAC was going to accelerate.
- BYU & FOA Morphology: BYU used a Wildforce board with medium size Xilinx 4K parts and one memory port per FPGA to implement the morphology part of the FOA (Focus Of Attention) algorithm stage (the first stage of the ATR which picks regions of interest for further processing). The FOA algorithms consists of three parts: adaptively quantizing the image, morphological processing to generate blobs, and a backend which reasons about the remaining blobs to determine if they are target-like. Most of the FOA processing time is in the first two parts. The backend has computations that have

significant amount of control flow and hence are not well suited to hardware implementation. The performance was good, but memory size limitations prevented the use of large input images, that was okay since boards with more memory would be coming later in the program.

- BYU & Chunky SLD: BYU used a Wildforce board to implement a more robust version of the SLD algorithm. The computations are similar to the baseline SLD algorithm but the image kernel is broken up into multiple chunks, and the results of the individual chunks are combined to get a score.

4.3.2 <u>1999</u>

- This year marked the transition to the more robust algorithm suite from Sandia for Joint STARS and for the SLAAC program. The more robust algorithm suite made SLD and Chunky SLD obsolete and replaced them with the Contamination Distribution Indexer (CDI). This also marked the start of transition to Generation 2 FPGAs (e.g. Xilinx Virtex family) since the more complex algorithms such as CDI needed more memory bandwidth, which even several external memory ports could not provide. The need for the on-chip memory banks had corresponded well with Xilinx's introduction of the Virtex family of FPGAs. This year for the demo, the system had to rely solely on embedded HPC components to meet the performance goal since no embedded second generation hardware was ready. Luckily, faster CPUs were available and the implementation of the more robust algorithms was drastically improved.
- BYU & FOA Adaptive Quantization + Morphology: This year BYU started designing hardware for the first part of the FOA algorithm which was the adaptive quantization (called superquant). BYU also worked hard on designing tools for rapid recompilation and debugging of designs. BYU also ported the design to the SLAAC-1 board.
- BYU CDI: BYU started the implementation of the CDI algorithm. The initial goal was to get the most computational part (CDI 2X) of the algorithm implemented first (~85% of the algorithm runtime). They designed to the WildSTAR board since that was the only board available at the time that had the Xilinx Virtex. This group had an extra tough time since they were the first on the SLAAC team to start designing to the Virtex part so they had to deal with all the design tool changes and bugs that weren't flushed out yet.

4.3.3 <u>2000</u>

- This year marked the migration to the SLAAC-1V. The SLAAC-1V most importantly had the Xilinx Virtex parts and their much needed on-chip memory blocks. The SLAAC-1V was basically the SLAAC-1 but with the Xilinx 4K parts replaced with Xilinx Virtex parts. The SLAAC-1 and SLAAC-1V have several advantages over any available FPGA boards at the time that were crucial to the success of the SLAAC program. The multiple external memory ports is desperately needed for high performance, one of the primary goals of this program. Also, often overlooked features such as significant hardware support for debugging and available source code and models so that special software tools can be developed to ease the complex design process.
- BYU & FOA Adaptive Quantization + Morphology: BYU started to migrate the design to the SLAAC-1V.

- BYU & CDI: BYU migrated the design from the WildSTAR board to the SLAAC-1V. Yet again this team lead the way in flushing out bugs in design tools. The performance was quite impressive, greater than 40 times the performance of a PowerPC. After the CDI 2X port to the SLAAC-1V was complete, BYU started to add the additional processing stages of the Contamination Distribution Indexer (CDI) algorithm so that less CPU pre- and post- processing was required. The stages include some image preprocessing, CDI 2X algorithm computation, followed by CDI 1X algorithm computation. The CDI 1X computation is the same as the CDI 2X computation except that significantly less templates must be correlated, the search region is significantly smaller, and there are about four times more pixels to correlate.

4.3.4 <u>2001</u>

- This year marked the attempt to complete the designs. It also marked the year that the SAR ATR system migrated from 6U VME to ruggedized SMP boxes for the less demanding environments. Besides being a factor of 5 or more cheaper than the 6U VME multicomputers, the ruggedized SMP box has PCI slots with which a SLAAC-1V or other PCI ACS boards could be placed. A SLAAC-1V board was put in the SAR ATR that was reinstalled on the updated Joint STARS T-3 aircraft. Unfortunately, the design was not complete in time for the test flight in June, so the system ran without ACS acceleration.
- BYU FOA Adaptive Quantization + Morphology: This year BYU basically completed the port to the SLAAC-1V. However, the designs were separate, hence would require two boards and extra bandwidth and CPU overhead to make them work together. The rest of the year was spent merging the designs to fit on a single SLAAC-1V board.
- BYU CDI: BYU continued to work on a single SLAAC-1V board implementation that did preprocessing, CDI 2X and CDI 1X.

4.3.5 <u>2002</u>

- This year marked the availability of “6 million gate” Xilinx Virtex-II parts. Virtex-II parts were used to make a second generation SLAAC-1V called Osiris. This is really a generation 2.5 device, not quite the generation 3 devicethat was envisioned in 1995 before the start of the project. Although the third of the three postulated generation 3 parts has been announced by Xilinx (Virtex-II Pro part with PowerPC processor cores), it is too late for the SLAAC project. Three things conspired to make the SAR ATR team not use the Osiris board: closeness to the end of the project, funding running very low, the port to the Osiris board would be basically be engineering and not stretching technology. Hence we concentrated on flushing out an final bugs in what we had for the SLAAC-1V and using that for the final laboratory demonstration.
- BYU FOA Adaptive Quantization + Morphology: BYU made some minor bugs fixes to the design and completed the task. The design was tested and validated at Sandia.
- BYU CDI: BYU made bug fixes and completed the task. The design was tested and validated at Sandia. The design made the algorithm run 10 times faster than without the ACS acceleration, that includes the overhead of the CPU doing some pre and post processing and module message handling. The design was integrated into the SAR ATR

system and end-to-end tests were performed. The goal of the tests were 2 Mpixels/sec for 30 target configurations while requiring only 1 cubic foot. We initially used a several year old dual processor Sun Ultra 60 with one SLAAC-1V and achieved 1.09 Mpixels/sec. A speedup of 3.4, which is quite good considering only the most computational intensive stage of the three stage system was being accelerated by ACS. The Sun Ultra 60 has a volume of 1.5 cu. ft. which is greater than the 1 cu. ft. goal but that was expected since there was no embedded version of the SLAAC-1V. We had let the ACS program manager know early in 2000 that we would not meet the volume goal but would be close. Peculiarities in the PCI bus of Sun Blade 1000 prevented us from using this faster machine, however predictions indicate that using a Sun Blade 1000 would have yielded 2.5 Mpixel/sec with SLAAC-1V acceleration which would have exceeded the performance goal of 2 Mpixels/sec.

4.4 Results

4.4.1 First Generation ACS

The start of the development of the first generation of ACS components was a disaster. Our intention was to use a four-port 6U VME network being developed by Myricom, Inc. with gigabit LANai network interface processors and develop FPGA-based processing elements as daughter card modules. However, Myricom did not deliver for so long that the team had to take another course of action, which was to develop a PCI board with multiple large Xilinx 4000s each with multiple memory ports. This became the SLAAC-1 board. A board with multiple memory ports per FPGA was crucial to the performance needs of the applications. Also complete access to the models and driver code was crucial to the development of the tools that were to be used by the SLAAC team. ISI-East did a great job of developing a board and software drivers and saved the project. A second board was also designed based on the PCI board but for the embedded version, this became the SLAAC-2 board. In hind sight, this was mistake number two. Too much effort and money was spent on developing a first generation embedded board when we knew performance of ACS would not be really compelling until we started using at least the second generation. This especially hurt since it prevented earlier development of the second generation researchers board (SLAAC-1V). In addition, because of the delay due to Myricom and the difficulties of developing an embedded board, the need for a first generation embedded board for SAR ATR was past by the time the SLAAC-2 was available since the algorithms had migrated to the more demanding robust algorithm suite which would not fit on the first generation FPGAs.

While the SLAAC-1 board was being developed, Sandia worked with Myricom to implement the SLD algorithm in an AT&T Orca FPGA on custom daughtercards that plugged into a custom Myricom switch board. This board was used to meet the 1998 performance goal. Myricom was able to do this since it was a very slight modification to the layout of existing products.

4.4.2 Second Generation ACS

The more complex algorithms such as CDI needed more memory bandwidth, which even several external memory ports could not provide. As predicted the Xilinx 4K like parts with on-chip block RAMs and multiplier support became available as the Xilinx Virtex parts. Initially the there was no SLAAC board available that had Virtex parts so we had to start the initial design with Annapolis Systems WildSTAR product until the SLAAC-1V became available. The SLAAC-1V was needed since it had multiple memory ports per FPGA, sophisticated debugging support, and complete access to the models and drivers for interfacing to the design tools and debuggers. The FOA and CDI algorithms were mapped to the SLAAC-1V, tested and verified.

The SAR ATR system migrated from 6U VME to ruggedized SMP boxes for the less demanding environments. Besides being a factor of 5 or more cheaper than the 6U VME multicomputers, the ruggedized SMP box has PCI slots with which a SLAAC-1V or other PCI ACS boards could be placed. A SLAAC-1V board was put in the SAR ATR that was reinstalled on the updated Joint STARS T-3 aircraft in 2001. Unfortunately, the design was not complete in time for the test flight in June 2001, so the system ran without ACS acceleration.

The CDI implementation on the SLAAC-1V made the algorithm run 10 times faster than without the ACS acceleration, that includes the overhead of the CPU doing some pre and post processing and module message handling. The design was integrated into the SAR ATR system and end-to-end tests were performed. The goal of the tests was 2 Mpixels/sec for 30 target configurations while requiring only 1 cft. We initially used a several year old dual processor Sun Ultra 60 with one SLAAC-1V and achieved 1.09 Mpixels/sec. A speedup of 3.4, which is quite good considering only the most computational intensive stage of three stages was being accelerated by ACS. The Sun Ultra 60 has a volume of 1.5 cft. which is greater than the 1 cft. goal but that was expected since there was no embedded version of the SLAAC-1V. The ACS program manager knew early in 2000 that we would not meet the volume goal but would be close. A documented Sun Blade 1000 PCI bus bug prevented us from using this faster machine with our PCI hardware. Projecting from measured Sun Blade 1000 performance for the unaccelerated software-only ATR modules and measured performance of the hardware-accelerated ATR modules on Sun Ultra 60 with SLAAC-1V, the performance of the Blade 1000 end-to-end system was 2.5 Mpixel/sec, which exceeded the goal of 2 Mpixels/sec. However, the PCI interface bug prevented a final demonstration on Sun Blade 1000 and we had to demonstrate on the slower Sun Ultra 60 at only 1.09 Mpixels/sec.

4.4.3 Third Generation ACS

The true third generation ACS, never occurred for the SLAAC program. A generation 2.5 of ACS board was developed (Osiris) but was never employed to tackle the SAR ATR challenge because of the factors explained earlier. The lack of a true third generation ACS board wasn't the fault of the SLAAC team. None of the three candidate third generation components: clock-by clock reprogrammable FPGAs, tightly coupled processing core, FPGAs and memories; and very large second generation FPGAs with on-chip microprocessors came into being early enough to be used. Large high performance clock-by-clock reprogrammable chips with on-chip block RAMs do not exist and the paradigm did not end up matching well with the SAR ATR algorithms. The second of the third generation candidate, designs which feature tightly coupled processing core with FPGA resources and memories were cancelled. The very large second generation FPGAs with on-chip microprocessors (although announced much earlier) are just now becoming available and being supported by the design tools from their companies.

5 FINAL STATUS REPORT ON THE SAR/ATR FOA ALGORITHM

Dr. Brad Hutchings
Brigham Young University

5.1 Introduction

The Focus of Attention (FOA) algorithms are used by Sandia National Laboratories as a preprocessing stage to their Automatic Target Recognition engine. FOA is used to process synthetic aperture radar (SAR) images with the goal of finding regions in the image which contain likely targets. The exact FOA algorithm used is dependent on the type and size of targets as well as the radar system generating the images. Weather conditions can also cause changes to be made to the algorithm. The goal of FOA is to reduce the size of the data set for the more computationally complex stages of ATR which identify the existence and type of targets.

Because of its demanding computational requirements, FOA was originally implemented on a specialized morphology computer, called the CYTO computer, manufactured by the Environmental Research Institute of Michigan (ERIM). The CYTO computer was constructed using custom ASICs to create a semi-programmable, image-processing pipeline computer. The CYTO computer could be programmed to perform a wide variety of image morphology operations using an arcane language known as C4PL (CYTO Portable Parallel Picture Processing Language). An FOA script then is implemented as a list of C4PL operators performed in sequence. Due to improvements in IC fabrication, the CYTO computer quickly became obsolete and has been replaced by a software implementation. However, C4PL outlived the CYTO computer and is still in use as the programming (or scripting) language used to describe FOA applications.

The current software implementation by Sandia National Laboratories is a collection of optimized C functions which emulate the operation of a subset of C4PL instructions. The Sandia software reads the FOA script and chains together the operations specified in the script. Due to advances in microprocessors, this sequential software approach is many times faster than the original CYTO computer.

5.2 Technical Approach

Even though the software version of FOA was a vast improvement over the CYTO computer, it still did not reach the performance goals set forth by Sandia, and accelerating FOA with custom ASICs, as was done with the CYTO computer, is not an option because the final solution must use COTS (Commercial off-the-shelf) hardware. Sandia indicated that the following requirements must be met in order for the FPGA-based FOA implementation to be of interest.

1. Assume programmers/operators have no hardware experience. Image processing specialists must be able to write the algorithm with no circuit design skills.

2. Actual FOA algorithms used are classified. The algorithm must be specified and mapped to hardware without involving unclassified personnel.

3. The algorithms are to be specified in the C4PL programming language.

4. Because operating conditions may dictate an algorithm change, in-field modifications must be possible. The maximum compile time that can be tolerated is half a work day (4 hours).

5. Throughput must be an order of magnitude better than the current Sandia software implementation.

To meet these requirements, we mapped the design to Xilinx XC4000 family FPGAs and utilized a two-level compilation approach. The first stage of the compiler generates a structural design specific to a particular platform, which provides the generic hardware structure sufficiently large to perform the desired operations. The second phase customizes the generic pipeline to perform the specified C4PL operations. The first phase is time-consuming requiring up to 2 hours, but must only be performed once per board-type. The second phase (which will occur each time a new FOA script is written) requires only seconds.

5.3 Results

The final FOA compiler met or exceeded all of the original constraints: 1) No hardware knowledge is required. Users need only be familiar with C4PL. 2) Compile times for FOA scripts (stage 2) are measured in seconds. 3) The throughput of the FPGA-based FOA hardware is 10x the throughput of the existing Sandia system. The Xilinx tools reported that our final implementation would operate at 50 MHz. However, the memories on the Wildforce board we were using were limited to operating at 46 MHz, so the circuit was never tested faster than this.

Because the circuit is pipelined and capable of processing a new pixel every clock cycle, the 46 MHz circuit would have a peak throughput of 46 Megapixels per second. Average throughput, taking into account the control overhead is about 44 Megapixels per second. This throughput is the same no matter what FOA script is used. The software written by Sandia does not exhibit this same behavior. Because of the inherently sequential nature of the software implementation, the throughput is script dependent. On average, we measured the software implementation to get 4.3 Megapixels per second on typical length scripts using a G4 PowerPC operating at 450 MHz. This means that the hardware implementation has a 10X performance increase over the software implementation, for average size scripts. The FOA system has been delivered to Sandia and verified by Sandia to be functionally correct.

5.4 Related Publications

A paper is attached to this report entitled "An Application-Specific Compiler for High-Speed Binary Image Morphology". It was presented at the FCCM'01 conference held in Napa CA in April 2001, details the final work, and provides additional performance comparisons.

6 FINAL STATUS REPORT ON THE SAR/ATR CDI ALGORITHM

Dr. Michael Wirthlin
Brigham Young University

6.1 Introduction

The Contamination Distribution Indexer (CDI) is a new, advanced automatic target recognition (ATR) algorithm developed by Sandia National Laboratories. CDI is one of several algorithms used within Sandia's airborne Synthetic Aperture Radar (SAR) ATR system. This algorithm extends the capabilities of previous approaches by supporting camouflage concealment and deception scenarios and providing adaptive attenuation of radar images. While this algorithm is superior to previous approaches, it requires over ten times more computation. Details on this algorithm and its importance are described further in Sandia's final report.

6.2 Technical Approach

The CDI algorithm requires significantly more computation power and memory bandwidth than available with traditional programmable processors. The CDI algorithm was originally designed to execute on a multiprocessor embedded PowerPC platform. Although the PowerPC provides relatively high-performance in such a small form factor, it does not offer enough performance to meet the real-time computational requirements of the pattern recognition system. ACS technology is used to exploit custom functional units, control, and memory access patterns, which offers an efficient alternative to traditional programmable processors.

The goal of this project was to create an ACS solution for CDI that performs at 10 times the throughput of a quad PPC750 within the same volume (16,000 templates/sec throughput). The ACS solution for CDI was designed to operate on the SLAAC-1V configurable computing machine and uses the SLAAC device drivers and SLAAC API. To achieve these peformance goals, the technial approach to this problem involves a custom configurable circuit and host software that exploits the aggregate bandwidth of distributed memory (on and off chip), customized the datapath, and exploits the natural parallelism found in the algorithm.

Unlike many ACS applications, CDI is limited by memory bandwidth, not computation rate. To achieve the anticipated processing rate, the computing system must maintain a sustained memory throughput of over 5 Gbyte/sec. This memory bandwidth is achieved by exploiting the Block RAM memories found within the Virtex architecture. A large number of Block RAM memories are distributed throughout the system and customized to operate on CDI histogram tables. Further, the memory accesses are optimized to insure that valid data is retrieved/stored during each clock cycle.

In addition to the distributed memory, a custom datapath was created to insure that the data is processed as soon as it is accessed from memory. While this algorithm is not limited in performance by the computation, it is necessary for the datapath to keep up with the distributed memory - any datapath operations that stall the memory will reduce the algorithm performance. For this computation, simple adders, subtractors, and memories are employed to operate on CDI histogram tables.

The ACS implementation of CDI exploits the natural parallelism of the algorithm to provide superior performance. Rather than operating on a single pixel location at a time, the ACS approach replicates 15 search processors in parallel. Using 15 processors allows the ACS system to search an entire row of the search region simultaneously. While the use of parallel search processors increases the peak computation rate of the system, it is difficult to keep the processors operating with valid image data. The internal processing rate of the system soon outpaced the off-chip memory bandwidth. A major challenge faced by this project was to insure that the parallel processors are not stalled due to memory limitations. A complex memory addressing and caching interface was created to maximize the computation rate of the internal processors.

By exploiting distributed memory, custom datapath, and algorithm level parallelism, an ACS solution was able to provide over 40 times the throughput of a single embedded PowerPC processor (over 10 times a Quad PowerPC). The ACS solution achieves this result in spite of the fact that it operates at one fourth the clock rate of the PowerPC. This allows the ACS solution to perform the computation with significantly less energy than conventional programmable processors.

Additional technical details of the ACS CDI implementation can be found in the published FCCM paper and Steve Morrison's Master's thesis.

6.3 Milestone Summary

6.3.1 1999

- 3/1999 - The new CDI algorithm is introduced to BYU by Sandia National Labs. The ACS architecture of CDI is discussed.
- 12/1999- The CDI ACS implementation was designed in JHDL and mapped to the Annapolis Wildstar board. This design was delivered to Sandia for system integration.

6.3.2 2000

- 4/2000 - The results of the CDI algorithm and performance are published at the IEEE Conference on Field Programmable Gate Arrays for Custom Computing Machines (FCCM).
- 6/2000 - The ACS CDI implementation is delivered to Sandia on the SLAAC1V.
- 9/2000 - After initial CDI integration tests, Sandia determined that the CDI system should integrate both the 2X and 1X phases of the algorithm. While this will lower the throughput of the CDI system, it will significantly reduce the latency of the operation. The combined 1X/2X phases of the algorithm are presented by Sandia to BYU. This involved a complete redesign of the CDI system.

6.3.3 2001

- 6/2001 - Modified 1X/2X delivered to Sandia. The current system achieves a clock rate of 52 MHz.
- 11/2001- Errors in the CDI implementation are identified by Sandia. Sandia provides additional test vectors to BYU for verification.

6.3.4 <u>2002</u>

- 2/2002 - Modified CDI implementation delivered to Sandia. Modified version resolves the errors identified by Sandia and incorporates a number of enhancements.
- 4/2002 - Correct operation of CDI verified by Sandia.

7 FINAL STATUS REPORT ON THE SONAR BEAMFORMING CHALLENGE

Dr. Brent Nelson
Brigham Young University

7.1 Introduction

The SONAR challenge problem addressed was to investigate the use of ACS technology for performing real-time SONAR beamforming. Prior to the start of the effort, ACS technology had been demonstrated mainly in applications areas with the following characteristics:

1. small data elements - typically 1-8 bits wide.
2. simple arithmetic - mainly logical operations and additions/subtractions
3. minimal control
4. amenability to pipelined computation
5. large amounts of data parallelism

These characteristics had previously been identified by BYU researchers as a distillation of the conventional wisdom at the time regarding what constituted a suitable application domain for FPGA implementation. SONAR beamforming represented a new challenge area in that it didn't necessarily meet characteristics 1-3 above. The proposed work was to adopt it as a challenge problem and determine if and how ACS technology could accelerate SONAR processing.

Prior to the start of the project an unfunded feasibility study had been done by BYU researchers to determine the suitability of ACS technology for SONAR beamforming. The preliminary results were positive. The key feature identified in that study was the very high levels of parallelism which existed in the computation.

7.2 Technical Approach

From the beginning of the project, the "customer" of the research was the Naval Undersea Warfare Center (NUWC) in Newport R.I. Our NUWC point of contact was W. Robert Bernecky. Mr. Bernecky played an important role in the project by providing relevant SONAR kernels to the BYU research team for implementation.

The approach consisted of three parts. The first was to complete the feasibility study on the appropriateness of ACS technology for SONAR beamforming. The second was to work with the SLAAC platform team to ensure hardware support for the demonstrations we desired to complete. The third was to demonstrate a laboratory-based algorithm demonstration. Finally, an option to scale up the demonstration and deploy a system at NUWC to support the Sonar Grand Challenge was proposed (and eventually exercised).

The majority of the effort was spent on the third part (creating a laboratory demonstration) and the option. The technical plan was to begin with the design of a series of simple beamforming kernels. For each kernel a paper design was to be done. The result of each paper design would be an analysis of the algorithm and its structure and a set of candidate implementations. In all cases we anticipated more than one candidate implementation based on different options for parallelizing the computation. Selected kernels were then to be implemented using an HDL and simulated to get more detailed area and performance numbers. Of those, selected kernels were to be further mapped to hardware and demonstrated executing on ACS platforms. The culminating design was to be a full-size design suitable for use at NUWC for demonstrations and sea-test.

This work was to leverage other parts of the SLAAC project. Hardware implementations were to be completed on SLAAC research reference platforms. The work was also to leverage software tools produced elsewhere in the SLAAC project.

The primary metric adopted for judging the success of the project was the product of circuit area and computation time for the finished SONAR processing modules. Power was not a consideration.

7.3 Milestones and Results

In FY 1997 we completed the feasibility study, using as an example problem time delay beamforming on a large spherical sensor array. The core of the design was committed to HDL and simulations run. The remainder of the design was a paper design. The results of the feasibility study indicated that 10-50X speedups over programmable CPU's and DSP's was possible using ACS technology.

In FY 1998 the first hardware mapping of a beamforming design to ACS was done. It was demonstrated 7/98 to Dr. Jose Munoz, DARPA ACS PM at the time. The design was a simple range-focused frequency domain beamformer design for a linear towed array. The mapping was done to the Annapolis Wildforce platform. Due to limitations of the hardware platform used the demonstration consisted of a small number of beams and sensors. The significance of this mapping was that it validated the unique beamformer design developed by BYU researchers which performed part of the computation using complex polar coordinates and then converted to a complex rectangular coordinate system for the balance. This design is well documented in the attached papers.

Also, in FY 1998 a family of SONAR subcompilers and module generators were completed and used in the above design. The initial CAD flow used was based on VHDL. Complex module generators were created by writing C++ programs which generated the modules of interest as structural VHDL designs. Because the inclusion of placement constraints into VHDL was non-standard and inflexible, many of the building blocks used by the module-generated structural VHDL had been initially created in the Viewlogic schematic entry tool. In the end the flow consisted of VHDL code, C++ code, and Viewlogic schematics, all combined together by a structural VHDL design. Not surprisingly, this flow proved quite unwieldy. During 1998 the JHDL tool for runtime-reconfigurable circuits was coming on-line, having been developed under a separate DARPA-funded effort. It had all the characteristics of a module generation framework desired and so was adopted for the module generator portion of the SONAR design work. The initial modules created were for pipelined multiplication and pipelined CORDIC computations. Within months the entire SONAR design flow had been converted to the JHDL tool. The result was a superior design environment.

The SLAAC quarterly report for the period ending June 2000 reported the following milestone:

"On the sonar application, BYU completed and delivered the sonar design to NUWC for system integration and testing. This design is based on the SLAAC1 rev.B board. NUWC testing results have been very positive - the operational speed of the design is significantly higher than original estimates and the system operates as planned. Preparations are being made for a sea-test of the system in the coming two quarters."

This was the final SONAR design completed. It implemented a novel matched field algorithm developed by NUWC personnel. The algorithm was especially well suited to shallow water environments, allowed forward-sector beamforming, and localized the target to a 3D voxel of ocean water (existing system simply provide bearing and very limited range information). The initial matched field beamforming algorithm was unsuitable for ACS implementation due to its complexity. NUWC personnel, in collaboration with BYU researchers, devised a new two-stage approach. The first stage computed a coarse 3D localization estimate based on a k-w beamformer design. This was done to 900 different ocean voxel locations. The second stage interpolated each voxel estimate to 400 different subvoxel locations. The result was 360,000 "beams", each targeting a different ocean subvoxel. The physical ocean area of each subvoxel was 20m range x 2m depth x 2 degrees in bearing.

The first stage of the design fit onto the X0 FPGA on the reference platform. The X1 and X2 FPGA's contained four second-stage subvoxel beamformers each. As with the earlier demo, this design relied on the polar/rectangular converting frequency domain beamformer as its core computational element. The resulting SLAAC1 rev. B board performed at 50MHz and delivered the equivalent of between 4 and 5 billion arithmetic computations per second.

This design was shipped to NUWC during June 2000. NUWC has been using it since for computations on synthetic and recorded sea data in its testbed. No bugs were ever identified by NUWC and no changes were subsequently made to the design. NUWC is currently working under the DARPA-funded Robust Passive Sonar (RPS) program to move this design to new technology and to augment it for use in that program. A second task in the SONAR area was the investigation of floating-point algorithms and modules for use in adaptive SONAR processing. The major milestone associated with this task was the creation and release of a family of floating point modules for Virtex-II FPGA's in March 2002. This family includes modules for addition, subtraction, multiplication, division, and square root. The modules were coded in JHDL and are available at http://www.jhdl.org.

7.4 Related Publications

The papers attached to the report associated with SONAR include the following:

1. "FPGA-Based SONAR Processing" by Paul Graham and Brent Nelson. This paper was presented at the FPGA'98 conference held in Monterey CA in February 1998. It essentially is the results document for the initial feasibility study undertaken in the early part of the project.

2. "GIGA OP DSP on FPGA" by Brad Hutchings and Brent Nelson. This paper was presented at ICASSP 2001 held in Salt Lake City in May 2001. Part of it describes results obtained in the ATR part of the SLAAC project and part of it focuses on the SONAR work.

3. "Configurable Computing and SONAR Processing - Architectures and Implementations" by Brent Nelson. This was presented at the ASILOMAR conference in November 2001. It reviews many of the previous SONAR results obtained under SLAAC and updates them for the latest Virtex-II technology.

8 FINAL STATUS REPORT ON THE AEGIS ELECTRONIC COUNTER MEASURES

Matt French - ISI

8.1 Introduction

The AEGIS Electronic Countermeasures Assessment (ECMA) Signal Processing Subsystem provides for detection and analysis of countermeasures and jamming signals in the AEGIS AN/SPY-1 shipboard phased array radar system. The ECMA equipment frame (i.e. cabinet) processes frequency channelized, log detected, and Analog-to-Digital converted signals from six radar processing channels, the Sum, Sidelobe, Alpha, Beta, Upper Guard, and Lower Guard channels. The ECMA processor assesses the level and type of ECM jamming encountered, determines the presence and the extent of clutter, and generates a jamming / clutter report that assists the radar computer in making the proper response to a jamming / clutter environment. Primary signal processing functions performed by the ECMA equipment include cover pulse detection, band analysis, jamming analysis, and clutter analysis.

The Advanced ECMA Processor is of interest to the Adaptive Computing initiative because it provides a meaningful test case for the applicability of the Adaptive Computing technology to a real fleet problem. As a result of efforts initiated in 1995 at Lockheed Martin Government Electronic Systems (GES) to upgrade the existing AN/SPY-1 ECMA processor, the U.S. Navy AEGIS Program Office, PMS-400B, has maintained a great deal of interest in leveraging ongoing DARPA programs to upgrade and simplify legacy processors such as ECMA.

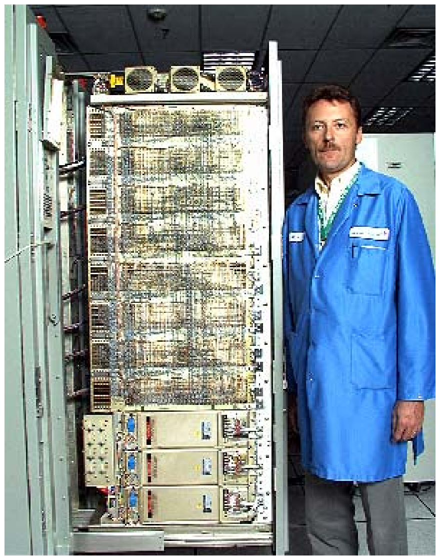


Figure 8-1 ECMA Cabinet

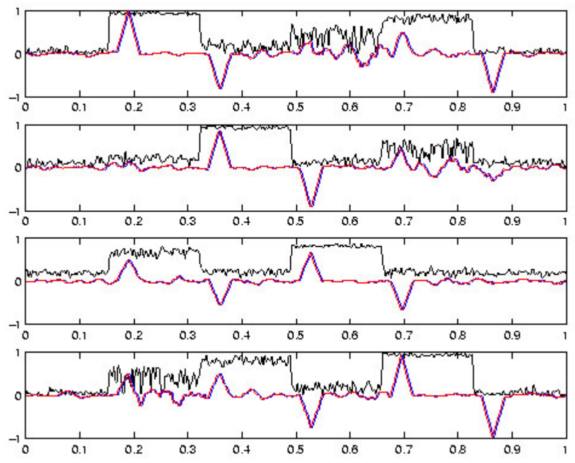


Figure 8-2 ECMA Module 2 Traces

8.2 Technical Approach

The goal of the ECMA Challenge Application was to duplicate the performance of the existing subsystem exactly. Success was to be measured by volumetric improvement. The ECMA equipment frame consists of approximately 35 different types of application specific electronic modules, filling an entire six-foot high, nineteen-inch wide equipment frame (i.e. cabinet). See the photo in Figure 8-1. The hardwired architecture is fixed - that is, it is not programmable, not flexible from a reconfiguration standpoint, and is not scaleable as requirements change. Modifications are costly and usually require expensive and long cycle module modifications to implement changes. The equipment cabinet is fully occupied with modules, there are no spare module slots left and there is no spare "glue logic" remaining in the equipment frame. Future, additional electronic countermeasure requirements cannot be met with the current equipment configuration. Control of the existing ECMA processor frame is provided from an external source, and as such the ECMA frame is the only electronic equipment frame in the AEGIS AN/SPY-1 D(V) signal processor that cannot be used or tested in a stand-alone configuration.

As described above, the ECMA algorithms require several types of signal processing functions, such as bit and byte manipulations, thresholding, magnitude comparisons, pulse length measurement, log average, up / down counting, and time delay, that may not lend themselves well to conventional digital signal processors. The ECMA processing algorithms provided an excellent exercise for implementation in the Adaptive Computing environment and valuable benchmark for the SLAAC. LMGES provided fourteen unclassified ECMA algorithm component specifications that had been captured from the AN/SPY-1 radar B-2 specification. ISI and LMGES implemented all fourteen components in VHDL and validated them on SLAAC-1 and then SLAAC-2 boards. Testbeds were deployed at LMGES and ISI. The final testbed system was demonstrated to the Navy at LMGES and to DARPA at ISI.

8.3 Milestone Summary

8.3.1 1998

- This challenge application was added to the SLAAC project statement of work. LMGES provided unclassified ECMA algorithm specifications captured from AN/SPY-1 radar B-2 specifications.
- LMGES provided unclassified test vectors for the algorithms.
- LMGES developed a demonstration test plan for SLAAC / ECMA.
- ISI developed ECMA modules in VHDL and tested them with these specifications.

8.3.2 1999

- ISI completed validation testing of fourteen ECMA modules on SLAAC-1 and delivered VHDL to LMGES.
- A SLAAC / ECMA Demonstration Test was provided for the AEGIS Program Office, PMS-400B, at the Navy Status Review in September 1999, on an Annapolis Microsystems, Inc. WildStar VME module in Moorestown.

8.3.3 2000

- LMGES was funded by Navy PEO TSC office to integrate ACS technology into Navy production systems.

- A second SLAAC / ECMA Demonstration Test was provided for the DARPA ACS Program Manager at the SLAAC Workshop in March 2000. The Demonstration consisted of a Real-Time Demonstration of ECMA functions on a SLAAC-1 module and a non-Real-Time display demonstration on a SLAAC-2 module with the display GUI running on an external UNIX workstation.
- A SLAAC-2 VME board was delivered to LMGES for integration testing.
- A SLAAC-2 VME board was delivered to Integrated Sensors Inc. for HRT-Express software integration.

8.4 Results

In the ECMA challenge, the goal was to achieve a volumetric reduction with identical functionality. LMGES identified fourteen major modules for the SLAAC developers to target. In 1999, all fourteen modules were implemented on a SLAAC-2 VME board achieving a 95% volume reduction with identical waveform outputs. In Figure 8-2, the black trace represents the input signal, the red trace the ECMA baseline, and the blue trace the SLAAC ACS implementation. The system was demonstrated to the Navy in 2000 and LMGES has successfully transitioned ACS technology to the Navy PEO-TSC TI and Navy PEO-TSC Production programs, as shown in Figure 8-3.

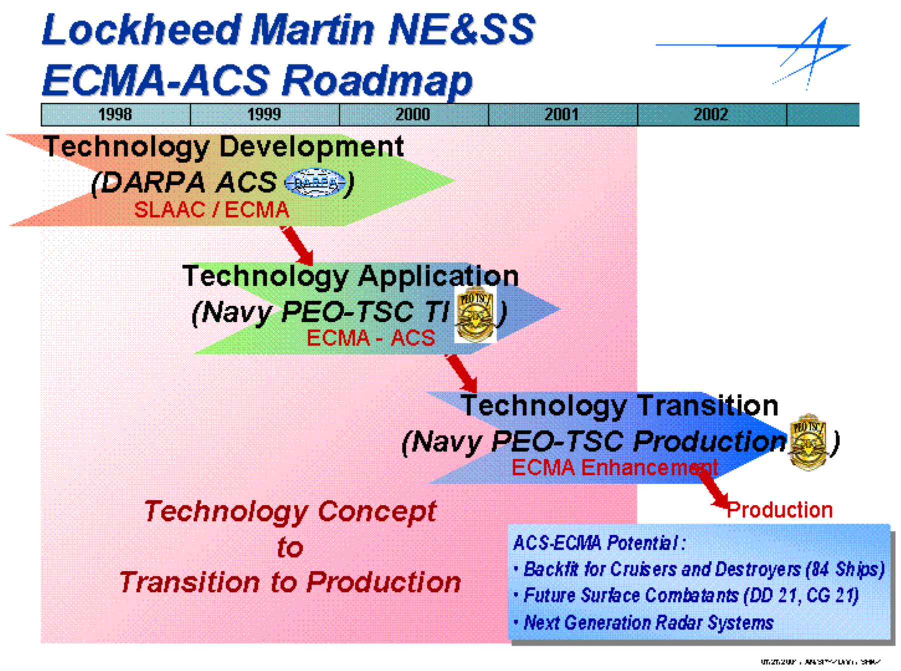


Figure 8-3 ECMA Navy Transition Path

9 FINAL STATUS REPORT ON THE WIDE BAND RF CHALLENGE

Dr. Maya Gokhale, Dr. Kevin McCabe
Los Alamos National Laboratory

9.1 Introduction

Los Alamos National Laboratory (LANL) Nonproliferation and International Security Division (NIS) provides technology to a wide range of DoE and DoD agencies. NIS supports a research program called Deployable Adaptive Processing Systems (DAPS) that seeks advanced signal-detection and feature-recognition techniques for RF and multi-dimensional image sensor projects such as Aldebaran/ Capella/ Cibola/ Bellatrix, Caliope, MTI, HIRIS/VIPER, SHS, and RULLI, as well as other government-sponsored initiatives. Our role in SLAAC has been to direct and validate DARPA technology via technology insertions into ongoing DoD programs. In 1999 Dr. Maya Gokhale joined LANL and brought her existing ACS funded Streams-C tool development program. The ACS program identified the Key Technological Challenges to reconfigurable computing. LANL supported SLAAC approaches to these challenges in the following ways:

- Building Blocks Inadequate - LANL drove the specification of new configurable computing modules and boards and supported their implementation. LANL developed and described new system-level heterogeneous architectures necessary for our challenge applications.
- Tough to program - Streams-C is a hardware/ software co-design compiler that supports automatic generation and scheduling of hardware and software processes via a streaming data model. It is the first open source C-to-Hardware compiler.
- Minimal runtime support - LANL developed a machine learning based image processing system that used automatic runtime configuration at the datapath level.
- Technology immature/risky - LANL has supported commercial suppliers to DoD by transferring LANL technology to Catalina Research Inc, which resulted in their Chameleon product line. Although Streams-C is not inherently hardware specific, our initial release targets Annapolis Microsystems's Firebird. LANL has developed applications for that board. Both companies have seen an increased customer base as a result.
- Problem formulation - LANL promoted and developed in concert with UCLA and USC/ISI a generic Digital Receiver Library of FPGAs components previously considered to be in the ASIC domain. LANL presented and developed new algorithmic approaches to multi-spectral and hyper-spectral image processing with support from USC/ISI.

9.2 Background

9.2.1 Capella Signals Analysis Workstation Accelerator

Capella was our first attempt to create a heterogeneous computing environment for UNIX workstation hosts, which improved interactive signal processing response times by at least 20X. At program start, Government-owned signal-processing packages required 4 hours to perform a rough search for signals of interest (SOI) on a 1 GByte file, using a Sparc20 mainframe. Capella demonstrated the execution in a few minutes to 10 minutes depending on the exact problem, using a VME/VXI crate co-processor. The performance of the system depended on using high-speed systolic data path connections. Capella was successfully delivered to the customer in support of national security and continues to evolve.

9.2.2 Bellatrix – Accelerated RF Signal Compression

Bellatrix is a proof of concept delivery program to a major USAF aircraft platform, which provides a path from basic systems R&D to operational deployment within the same program. An extremely challenging and inflexible mission goal was identified: Provide a real-time 100 Megasample/sec, 12 bit input resolution heterogeneous system that performs lossy compression on wideband RF signals, while meeting analyst expectations for no loss of important signal information. The initial goal is ~4X compression to a 50 MB/sec RAID however future applications will require much higher ratios. One such requirement would be to expand the input rate to 500 MHz, indicating a 1200 MB/sec input rate, while maintaining the 50 MB/sec output rate. Another application would maintain a 40 MHz, 100 MB/s input rate, but would reduce the output rate to ~3 MB/s. Either application requires algorithms that can provide 30X data compression with minimal loss of the desired signal information.

9.2.3 Cibola Ground Demo/Cibola Flight Experiment

The Cibola Ground Demo program seeks to prove the feasibility of a digital receiver deployed in space using ACS components. After a successful demonstration, the program has entered a second phase called the Cibola Flight Experiment, an effort to develop a space flight demonstration. The Cibola ground demo is an accelerated workstation that uses common elements with the Bellatrix and Capella programs. For example, Bellatrix uses the same channelized detector used in Cibola for a lossy time based compression scheme otherwise known as burst digitization. The Cibola Flight Experiment is a LANL design space qualified reconfigurable computer using Xilinx Virtex FPGAs. Cibola uses the components in the Digital Receiver Library.

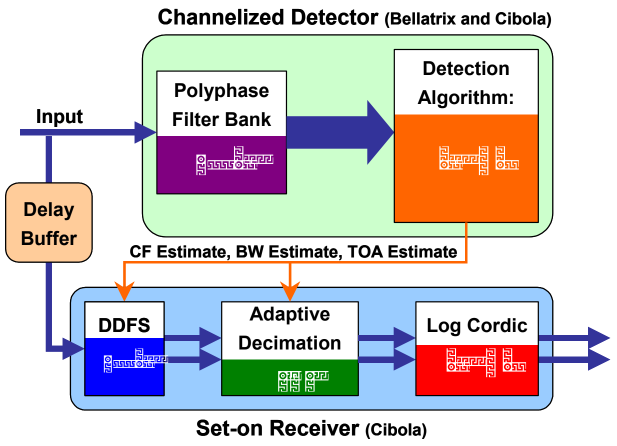


Figure 9-1 Digital Receiver Block Diagram

9.3 Technical Approach

LANL defined 6 specific wideband RF processing tasks that address new requirements within the UWB challenge problem. These tasks are classic ASIC functions and their implementation will demonstrate evolution from ASICs to FPGAs. This is in contrast to the evolution from C code to FPGAs of wideband RF algorithms being performed at Sanders and LANL. These tasks have been chosen for their enormous impact on problems being addressed by our DoD customers and have been validated by them. Together they form a Digital Receiver Library for RCC that has general application and interest in the Wideband RF community. Some of these tasks push the limits of current FPGA technology. For this reason we are looking at Xilinx Virtex 1000 FPGAs as a minimum. These tasks are:

- 32 Channel Polyphase Filter
- 32 channel FFT
- Statistical Detection
- Direct Digital Frequency Synthesizer
- Adaptive Filter (Sub-band Tuner)
- Log Cordic

These elements comprise a complete digital receiver architecture. DARPA technology is the Polyphase Filter and Frequency Synthesizer developed at UCLA and the Adaptive Filter developed at USC/ISI. LANL programmatic contributions are the 32 channel FFT and Log Cordic algorithm, contracted to Andraka Associates under the Cibola program, and the Statistical Detector, developed at LANL under the Aldeberan Program.

9.3.1 <u>32 Channel Polyphase Filter</u>

LANL had extensive interactions with Prof. John Villasenor's team at UCLA in defining a polyphase filter implementation. This algorithm has applications to both the Capella and Bellatrix projects. The performance metric was to implement a filter with specified characteristics at 100Msamples/second. The implementation was based on the basic SLAAC-1 PE architecture allowing for 4 16 bit input buses and 4 16 bit output buses. The polyphase structure is highly parallel and efficient, ideally suited for RCC. The filter bank consists of a decimator at the input, feeding a bank of 16 filters with 4 taps and 2 outputs each, optimized for symmetry. The outputs of the filter bank feed a commutator that forces the data onto 4 lines that feed an FFT in a separate FPGA. System gate and frequency estimates were performed for current and anticipated hardware. The targets are the XC40150XV-09 used on the SLAAC-1, the EPF10K250AGC599-1 used on the LANL RCA-2, and the Virtex XCV1000-4-BG560 used on SLAAC-1V, Catalina Research Chameleon, and Annapolis Microsystems Wildstar. UCLA completed a functional verification after which LANL performed simulations for performance verification.

The polyphase filter bank is a multi-rate polyphase filter structure combined with a Fast Fourier Transform (FFT) designed to split an input signal into various frequency subbands. The polyphase filter portion of the structure is based on a prototype baseband lowpass Finite Impulse Response (FIR) filter with symmetric coefficients, i.e., the first N/2 and the last N/2 coefficients are the same, albeit in reverse order. The remaining filters of the filter bank are frequency shifted versions of the prototype. First a prototype filter, h0[n], with the desired filter parameters is designed. Polyphase filters, pk[n], are expressed in terms of the prototype filter, pk[n] = h0[k +

lM] k = 0..K-1, l = 0..L-1, (where N=128 is the length of our FIR prototype,) M is the number of channels (M = 32), and L is the length of the individual polyphase filters, (L = N/M = 4). The FFT is used following the polyphase filtering structure to provide the frequency shifts for the various channels.

LANL had preliminary discussions with UCLA on a follow-on algorithm, a filter that can run at a programmable rate. UCLA coded a VHDL behavioral model but a synthesizable model was not implemented due to resource constraints and program timing. They designed a complete time-frequency architecture similar to the polyphase filter bank based on a LANL Matlab model. This time, the number of channels varied, as many as 1024, instead of the fixed 32 channels in the first implementation. Likewise, the amount of decimation was allowed to vary so that the structure is no longer critically sampled but over sampled in many instances.

9.3.2 Direct Digital Frequency Synthesizer

LANL specified a Direct Digital Frequency Synthesizer to Prof. Rajeev Jain's team at UCLA. They delivered a VHDL design that exceeded all design parameters. The design synthesized frequencies to 12 bits at 100MS/second.

9.3.3 Adaptive Filter (Sub-band Tuner)

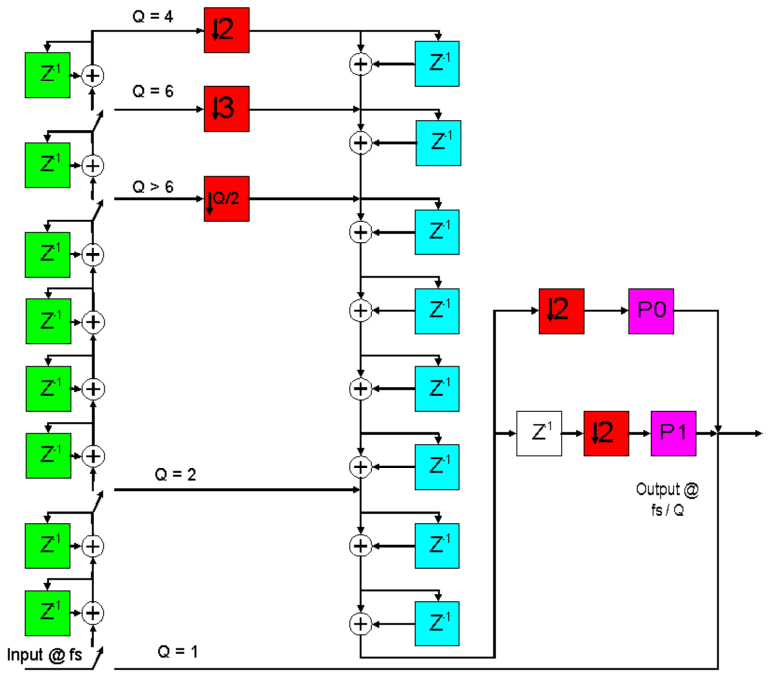


Figure 9-2 Hogenauer CIC/Polyphase Filters

LANL specified an Adaptive Filter to USC/ISI. LANL looked at two approaches. One approach is a Cascade filter and the other approach is a CIC filter, as described by Hogenauer, followed by a polyphase filter. The CIC filter offered several advantages. Among them, a greater freedom in decimation rates and an efficient hardware implementation. A Matlab model was created by LANL and the model and bit precise test benches were given to USC/ISI. Matt French of USC/ISI coded a synthesizable implementation for the SLAAC-1V. The filter is capable of

50Msamples/sec. A variation of the filter with a decimation by 2 filter on the front end resulted in a filter that is capable of 100Msamples/sec. This filter was ported by LANL to the Catalina Research Chameleon as part of the test and integration plan for the Cibola Flight Experiment.

9.4 Results

The Digital Receiver Library was implemented as planned. This was an enabling technology for the Capella, Bellatrix and Cibola wideband RF programs at LANL and is the foundation for several programs on the near horizon. At program inception the ability of available FPGA technology to realize these DSP structures was unknown. Proposed architectures were based on engineering judgment and to some extent faith. Cappela led the way with early engineering studies but the first true delivery was by Bellatrix to the Air Force with a successful airborne flight demonstration in July 2000. In a letter dated September 2000 from Captain Gary Jones, Project Engineer for the Combat Sent Program to Dr. Phillips, ACS Program Manager at that time, he states

"We understand that the enabling Bellatrix technology is based on FPGA based data processing and that the Adaptive Computing Systems program at DARPA has made a contribution via USC and UCLA of a wideband channelizer (polyphase filter), an essential component of the signal processing chain. The channelizer is the front end of statistical detection scheme to trigger wideband data acquisition for even weak wideband RF signals. We appreciate how our program to upgrade ELINT collection has been able to leverage off of DARPA/TTO/ACS technology."

The same technology used in Bellatrix was adapted to the Cibola Ground Demo, which met its goals in September 2000 of demonstrating a 100Msample/second statistical trigger in Xilinx Virtex technology suitable for space qualification. In Phase 2 of the Cibola program, called the Cibola Flight Demo, the Set-on Receiver was implemented and is still currently under testing and characterization. The Cibola Flight Demo was rated First in the Air Force SERB in the Fall of 2000. Cibola is still waiting for a flight opportunity at this time.

The strength of these programs depended on two key ACS payoffs.

- The ability to perform iterative development: This led to an early proof of concept and faster system development at a much lower risk than an ASIC solution.
- The ability to upgrade systems with rapidly advancing new FPGA technology.

9.5 Publications

Joseph Arrowood, Kevin McCabe, Kim Ruud, and Mark Dunham, "An Economical 40 MHz Universal Software Radio Using a Hybrid Approach," presented at High Performance Embedded Computing Workshop 2000, 20-22 September, 2000, Boston Mass.

10 FINAL STATUS REPORT ON THE GABOR FILTER BANK ALGORITHM

Dr. John Villasenor
University of California, Los Angeles

10.1 Introduction

In this section, UCLA's work on the Gabor filter, in cooperation with Los Alamos National Laboratory, is discussed in detail. UCLA worked on the design and hardware implementation of a Gabor filter bank. The system was designed to show the time-varying nature of the frequency components of a signal, something that the conventional Fourier transform could not show. The Gabor filter bank gave a time-frequency representation of signals by implementing a Gabor transform, which essentially consisted of multiple FFTs of a time-shifted and windowed signal. The filter bank accepted 8-bit inputs at 100Megasamples/sec, the number of window coefficients, the number of output channels, and the downsampling factor. The system output windowed signals, which were essentially the products of the signal inputs and the corresponding filter coefficient. See the following diagram.

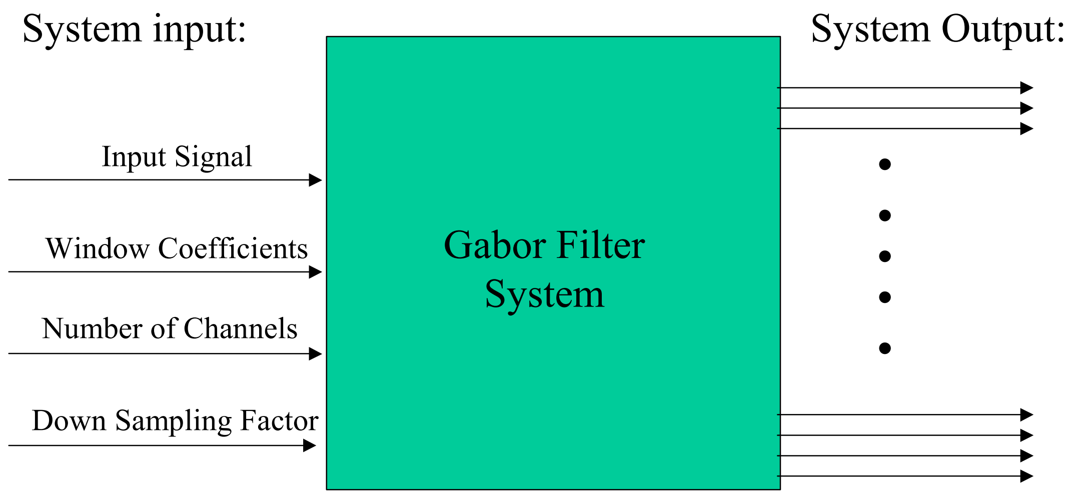


Figure 10-1 Gabor Filter System

Gaussian windows were used to window the input signal. As the signal came into the filter, the Gaussian window was multiplied to it to obtain a small chunk of the signal. The magnitude of the signal at the edges of the window was therefore lessened in magnitude. To reduce this problem, consecutive windows were overlapped so that the parts of the signal at the edges of the window were added and, thus, the change in magnitude was accounted for.

The Gabor filterbank was very adaptive. The number of window coefficients, and the number of output channels, varied between 128 and 1024 and were always a power of two. In addition, the windows could either have no overlap, 1/8 overlap, or 1/4 overlap. All the options were specified as the input for each run.

Because of its high adaptability, the system had several parts other than the actual filter to serve different needs. Specifically, it contained two commutators, one state machine, and the 128 filters that made up the filter bank. See the diagram below.

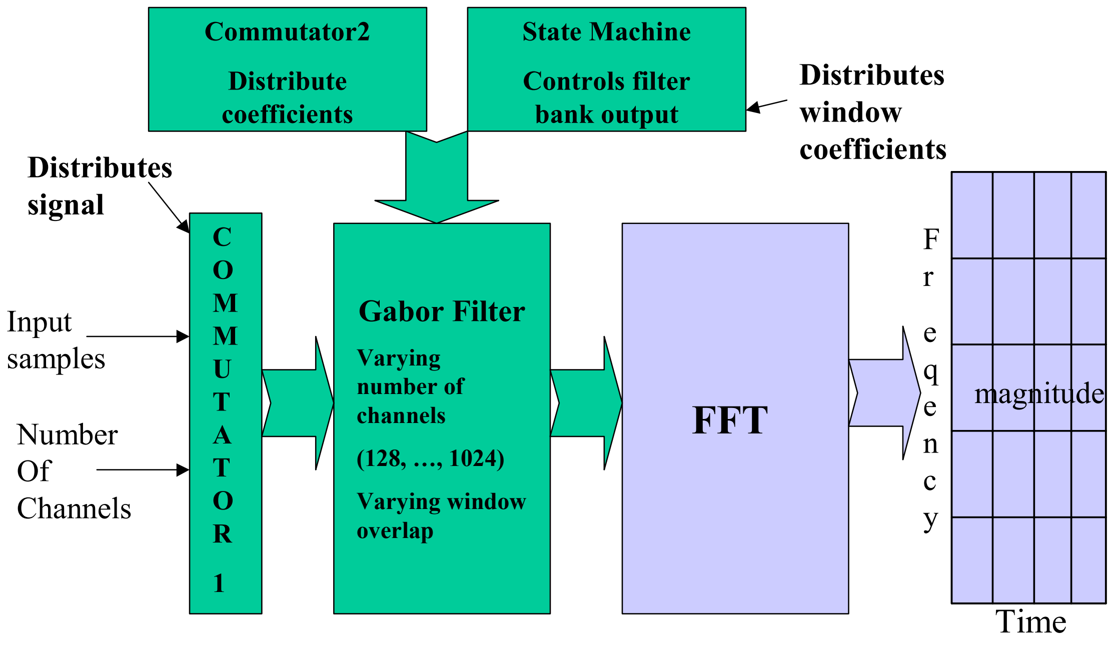


Figure 10-2 Gabor Filter Block Diagram

Commutator1 had two functions. On one hand, it accepted input signals, accumulated them, and output them to the filter at the appropriate time. The number of signals it output at a time depended on the number of filters used for the particular specification. On the other hand, the commutator also reversed the order of the signal every other time because of the way the window coefficients were placed in the filter banks. This will be explained in more detail below. Commutator2 distributed window coefficients according to the number of output channels and the number of window coefficients, since these two parameters determined how many filters in the filter bank would be used. The coefficients were distributed vertically rather than horizontally within each filter. See the diagram below.

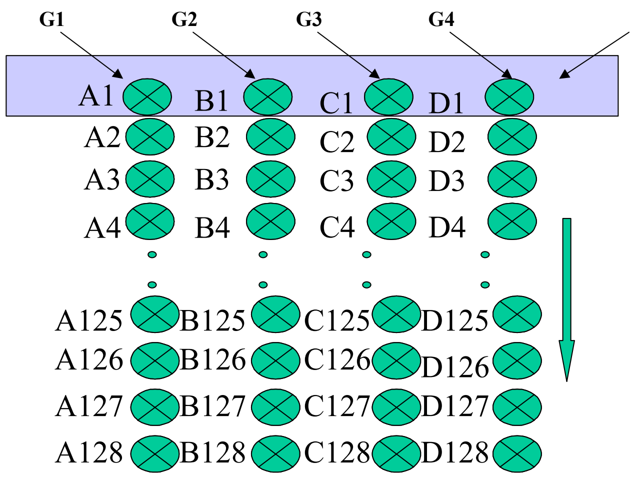


Figure 10-3 Commutator Structure

The filter bank consisted of 128 filters. Each filter contained four multipliers and ten registers. Each multiplier had signal as one input, and window coefficients as the other. The registers were memory elements that stored the results. Since the number of the outputs of the system was pre-defined, not all the results were output during the cycle in which they were computed. Registers were therefore used to hold them temporarily.

Finally, the state machine was the brain of the filter bank. It controlled the input and output of the registers (and hence the output of the system) in the filter bank by simply controlling when inputs were accepted into the filter and when outputs were to be generated.

Here is how the data flowed in the system. First, the input signals went to the commutators that downsampled signals and placed them in the appropriate order according to the filter arrangements. Meanwhile, coefficients were distributed into the filters by commutator2. Then the signals were loaded into the Gabor filters that took the product of the signals and their corresponding coefficients and stored them into the correct register. The state machine controlled the registers, and the results were released when the registers were told to do so.

Since the number of window coefficients could range up to1028 and each coefficient required a multiplier, optimization was critical in designing the system. The Gabor Filter bank took advantage of the symmetric Gaussian window and overlap of the windows to optimize system size and speed. Since the Gaussian window was symmetric, only half of the coefficients needed to be loaded into the system. This greatly reduced the size of the filters. Time optimization was achieved when overlap was required.

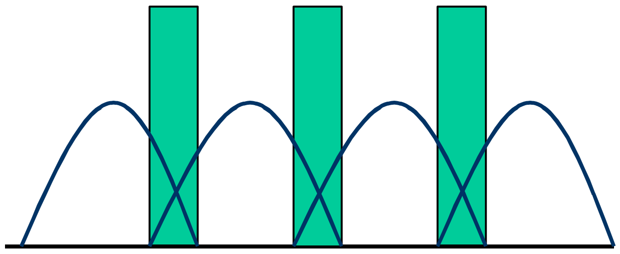


Figure 10-4 Gabor Filter Windowing

As shown in the diagram above, some signals were 'windowed' by the end of the first window and by the beginning of the second window. These two operations could be done at the same time as long as the results were stored and released at the correct time. The extra results from the second window were stored in the added registers in each filter and were released according to the state machine.

Although the Gabor filter bank was implemented with various optimization schemes, it was still very large due to the large number of multipliers. An interesting solution to this problem might have been to use re-configurable hardware. However, UCLA has not had an opportunity to explore this option further.

10.2 Milestone Summary

10.2.1 <u>1998</u>

- Under SLAAC, UCLA continued their investigation of the optical flow algorithm for the Los Alamos application at a "background" level. The optical flow algorithm was used to detect plumes in images. The algorithm looked at image intensities in 3 adjacent frames and returned velocity estimates based on this information. The memory bandwidth required to compute this information was great and represented a major challenge.

10.2.2 <u>1999</u>

- In early 1999, UCLA began algorithm work with LANL on polyphase filtering. This effort was in the study phase at that time. However, from an initial survey, UCLA was optimistic that the problem was well suited to FPGAs and that efficient mappings could be achieved.
- UCLA worked to develop a simulation model of the filter bank. In order to develop the model, the UCLA team needed to fully understand the various components and data flows within the system. The team focused on the design of the filter bank and the actual data output in each bin of the FFT. The filter specifications were provided in Dr. Fiore's paper.
- As far as the actual output of the system, a comprehensive understanding of the theory behind polyphase filter banks was necessary. UCLA did extensive work toward gaining this understanding, which included going through the equations involved to show that the

output bins of the DFT represented bandpass filtered versions of the input signal. This provided a mathematical representation and justification of the polyphase structure.

- It was also crucial to understand how a polyphase implementation affected the computational nature of the circuit, and how this applied to a gate-array architecture. In general, implementing the sub-band extraction using a polyphase structure allowed a large reduction in the complexity of the filters needed. Also, the ability to parallelize the computation by sub-band allowed a reduction in clock speed by a factor of the number of sub-bands. Large, parallel structures were well suited to FPGAs, so UCLA was optimistic that efficient mappings into FPGA architectures would be possible.
- By mid-FY1999, UCLA implemented this polyphase Discrete Fourier Transform (DFT) structure in VHDL, targeting both the Altera Flex10K and Xilinx Virtex FPGAs. The intention was to provide a feasibility study of possible hardware platforms with respect to various filter and transform parameters. Among these parameters were the data and coefficient precisions, as well as the number of subbands needed. UCLA implemented the code such that these parameters were adaptive.
- Initial studies showed that there were some resource problems (fast carry chains) on the FLEX10K that would not allow full implementation of the filter bank on this chip. UCLA took advantage of the symmetry of linear filters and devised an optimized version of the polyphase bank that did fit the FLEX10K. Later, the feasibility question involved integrating the DFT with the filter bank on the same chip. There were other optimizations done at this stage as well that greatly reduced the size of the design, thereby increasing the number of viable options. In short, UCLA provided LANL with an extensive feasibility study based on various constraints within the system. This allowed for the delivery to the customer of an educated and realistic set of options with which to design the final system specifications.

10.2.3 <u>2000</u>

- At the beginning of 2000, the polyphase filter bank RTL VHDL models were simulated. A DSPCanvas model that could generate a variety of simulation test vectors for varying bit precisions was built and used for generating the test vectors. The simulated models were then placed and routed on a Virtex chip. They did not fit on an Altera Flex10K. A DSPCanvas model was built to compute the SNR and analyze the frequency response of the prototype low pass filter for varying input, accumulator and coefficient precisions. The results of the analyses were sent to LANL.
- In July and August 2000, through journal articles forwarded by LANL, preparations were made for the design of a windowing system that produces a signal's time and frequency components simultaneously. In September, UCLA worked on the hardware layout of the system, according to the project definition. Starting in October, the team wrote the behavioral description of the system with parameterized bit precision. This model was compared with the Matlab model from Los Alamos to finalize the bit-precision of the system.

10.2.4 <u>2001</u>

- During the beginning of FY2001, work continued on the Gabor filter system for the application supplied by Los Alamos. The Gabor system produced a time-frequency representation based on Gaussian windows. In the Los Alamos application, the window size could vary between 128 and 1024 coefficients. There was also flexibility with respect to the amount of overlap size relative to the window length. Most of the effort in early FY2000 was aimed at developing, implementing, and testing optimization schemes that would allow this very large design to occupy the lowest possible number of logic gates.
- Two specific optimizations were performed. The first was the division of coefficients. Since Gaussian coefficients were symmetric, the number of multipliers could be halved. To further reduce computation, the coefficients were further subdivided into groups of four for efficient input into the filters. The second was multiplier reuse. This was another optimization that was enabled by symmetric coefficients. A commutator was used to further reduce the number of multipliers.
- The three major building blocks to the VHDL part of the design were as follows:
    - 1. Filter: The filter performed the multiplications and stored the result in a temporary register. These registers were controlled by enable signals, which in turn were controlled by the state machines.
    - 2. State machines: Since there were different output requirements for different inputs, multiple state machines to deal with different situations were required.
    - 3. Commutator: To reuse multipliers, signals were queued with the commutator, which essentially reduced the input rate by a factor of four.
- By mid-FY2001, UCLA completed the three major building blocks (Filter, State Machine, and Commutator), and work began on the integration of the parts into one cohesive unit. In August 2001, UCLA submitted to LANL the completed VHDL for the Gabor filter.

10.3 Related Publications

[1] K.-N. Chia, H. J. Kim, S. Lansing, W. H. Mangione-Smith, and J. Villasenor, *"Configurable Computing Solutions for Automatic Target Recognition,"* IEEE Transactions on VLSI, vol. 6, pp.364-371, 1988.

[2] J. Jean, X. Liang, B. Drozd, K. Tomko, *"Accelerating an IR Automatic Target Recognition Application with FPGAs,"* IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.

[3] S. Crago, B. Schott, and R. Parker, *"SLAAC: A Distributed Architecture for Adaptive Computing,"* in IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998.

[4] B. Schott, C. Chen, S. Crago, J. Czarnaski, M. French, I. Hom, T. Tho, T. Valenti, *"Architectures for System-Level Applications of Adaptive*

Computing," IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.

11 FINAL STATUS REPORT ON THE HYPERSPECTRAL IMAGERY CHALLENGE

Dr. Maya Gokhale, Dr. Kevin McCabe
Los Alamos National Laboratory

11.1 Introduction

Los Alamos National Laboratory (LANL) Nonproliferation and International Security Division (NIS) provides technology to a wide range of DoE and DoD agencies. NIS supports a research program called Deployable Adaptive Processing Systems (DAPS) that seeks advanced signal-detection and feature-recognition techniques for RF and multi-dimensional image sensor projects such as Aldebaran/ Capella/ Cibola/ Bellatrix, Caliope, MTI, HIRIS/VIPER, SHS, and RULLI, as well as other government-sponsored initiatives. Our role in SLAAC has been to direct and validate DARPA technology via technology insertions into ongoing DoD programs. In 1999 Dr. Maya Gokhale joined LANL and brought her existing ACS funded Streams-C tool development program.The ACS program identified the Key Technological Challenges to reconfigurable computing. LANL supported SLAAC approaches to these challenges in the following ways:

- Building Blocks Inadequate - LANL drove the specification of new configurable computing modules and boards and supported their implementation. LANL developed and described new system-level heterogeneous architectures necessary for our challenge applications.
- Tough to program - Streams-C is a hardware/ software co-design compiler that supports automatic generation and scheduling of hardware and software processes via a streaming data model. It is the first open source C-to-Hardware compiler.
- Minimal runtime support - LANL developed a machine learning based image processing system that used automatic runtime configuration at the datapath level.
- Technology immature/risky - LANL has supported commercial suppliers to DoD by transferring LANL technology to Catalina Research Inc, which resulted in their Chameleon product line. Although Streams-C is not inherently hardware specific, our initial release targets Annapolis Microsystems's Firebird. LANL has developed applications for that board. Both companies have seen an increased customer base as a result.
- Problem formulation - LANL promoted and developed in concert with UCLA and USC/ISI a generic Digital Receiver Library of FPGAs components previously considered to be in the ASIC domain. LANL presented and developed new algorithmic approaches to multi-spectral and hyper-spectral image processing with support from USC/ISI.

11.2 Background

11.2.1 <u>Rapid Feature Identification Project (RFIP) and Accelerated Image Processor using Machine Learning (POOKA)</u>

The DAPS team demonstrated a simple genetic algorithm (GA) approach for multi-dimensional image processing (MDIP) to their sponsor. The motivation for LANL MDIP work is to be able to perform rapid broad area searches. LANL is continuing development of the GA through hardware acceleration. RFIP drives RCC development in two ways. GAs consist of repetitive trials of mutating sequences of simple operations (genes) linked to form algorithms (chromosomes). The chromosomes are graded against "truth" data and the least successful weeded out. Thus it can be seen that fast run time as well as run time reconfiguration are very desirable.

POOKA is a LANL hardware accelerated multi-spectral processing algorithm. It is an outgrowth of RFIP research, which is broadly termed evolvable hardware. Evolvable Hardware is a new design methodology that uses stochastic optimization techniques such as evolutionary algorithms or simulated annealing to explore traditionally intractable design spaces. POOKA uses this idea to find hardware solutions to broad area search problems with orders of magnitude greater throughput compared to conventional software solutions. POOKA employs run-time reconfiguration at the algorithm level.

11.2.2 Hyperspectral Infrared Imaging System (HIRIS)

HIRIS is an experimental airborne Fourier Transform Spectrometer that collects hyperspectral data in which we are looking for certain spectral signatures by evaluating a set of matched filters. The matched filters must be "calibrated" constantly due to changing factors such as time of day, look direction, and atmospheric conditions to name a few. HIRIS produces data at roughly 67MB/s. and acquires an image cube in about 3 seconds. The data is stored and later processed on the ground. It takes 3.75 minutes to process 3 seconds of data on a high-end multiprocessor server. There is a critical need for real-time feedback at the sensor for it to become a deployed asset. Our approaches have been to focus on 1) accelerating the existing data processing algorithm and 2) formulate new algorithms that are more suitable to hardware acceleration.

11.3 Technical Approach

11.3.1 Rapid Feature Identification Project (RFIP) and Accelerated Image Processor using Machine Learning (POOKA)

LANL has developed an architecture to accommodate rapid partial reconfiguration required by evolvable algorithms. This architecture was published at both the 1999 and 2001 NASA/DoD workshops on Evolvable Hardware. Our rapidly reconfigurable architecture consists of partially reconfigurable functions within a partially reconfigurable interconnect structure. The partial reconfiguration is achieved through application specific control registers defined on the FPGA as opposed to a full bit stream reconfiguration.

Most real-world data-sets in image and signal processing are subject to unknown variations in the environment and sensor, which makes solutions difficult to find. Usually an ideal sensory input is assumed, which has been corrupted by a particular type of noise. By filtering the corrupted signal, the noise can be suppressed and the ideal signal recovered. Often assumptions are made, so that an optimal filter can be found. For example, if the noise is assumed Gaussian, an optimal linear filter can be found. When noise is non-Gaussian, nonlinear filters can be more effective. A natural approach to deal with the combination of Gaussian and Non-Gaussian noise, found in most practical problems, is to define a hybrid system that incorporates both linear and non-linear filtering behavior. This approach has been shown to be very successful for practical problems, and motivated the design of our parameterized network node for optimal image filtering. The node as illustrated in Figure 11-1 is based on hybrid Morphological/Linear building blocks.

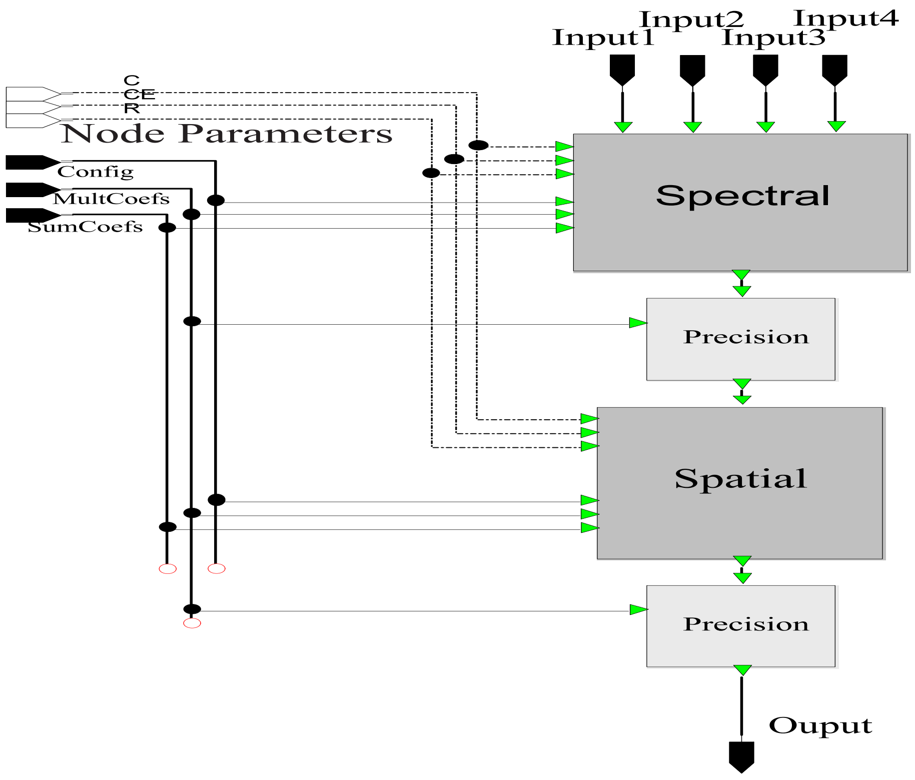


Figure 11-1 A Parameterized Node for Optimal Linear/Non-Linear Image Processing

The node has four inputs and one output and implements both spectral and spatial processing. The Spectral component means multiple image channels can be combined. These could be different spectral channels of a multi-spectral imaging sensor such as Red, Green, Blue and Near Infrared. Inputs could also be potentially from different sensors. A future extension, is to receive input from different moments in time.

Modern sensors are now being produced with increasing spatial resolution. To exploit these advances in sensor technology the node then performs spatial filtering using a small local neighborhood. By setting the parameters appropriately many powerful linear and non-linear image-processing algorithms can be implemented. Some of these are illustrated in Figure 11-2.

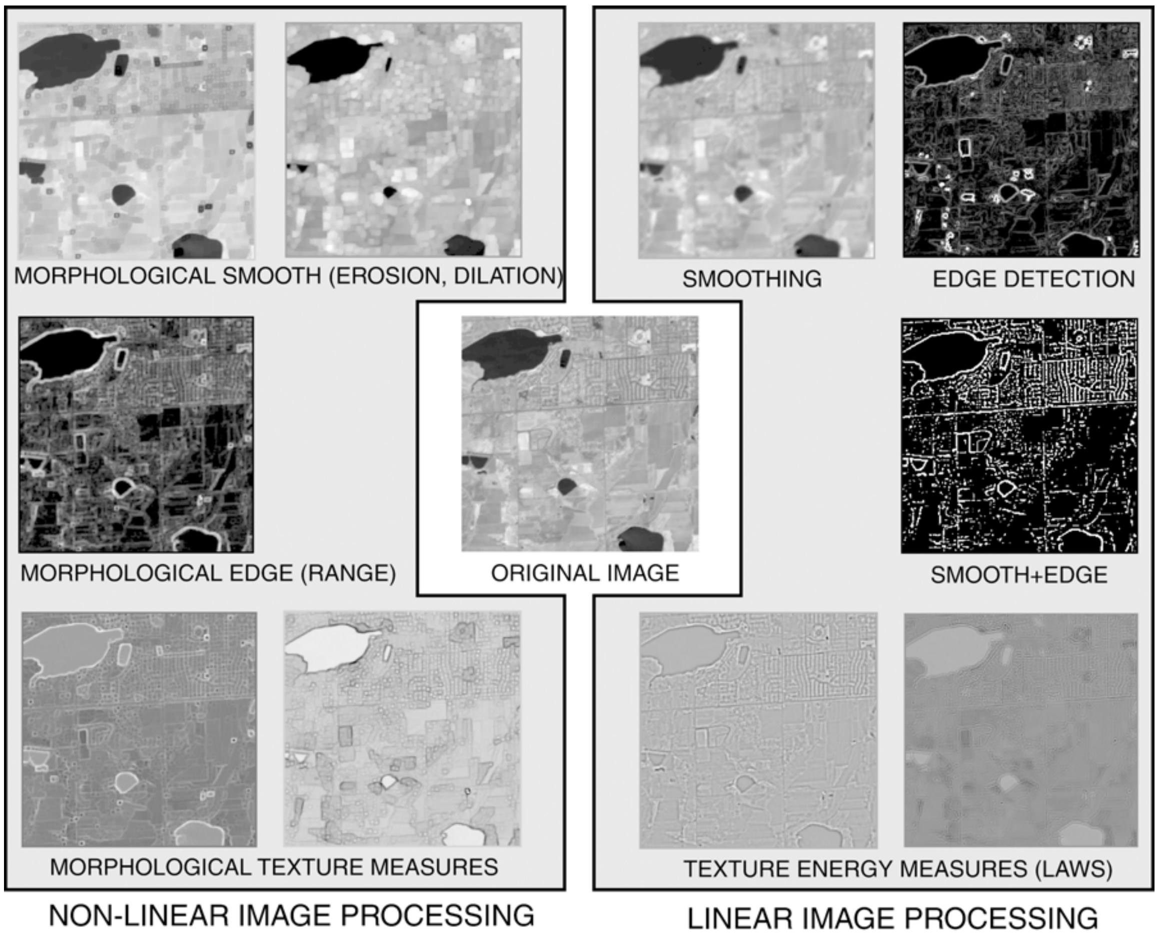

MORPHOLOGICAL SMOOTH (EROSION, DILATION)
SMOOTHING
EDGE DETECTION
MORPHOLOGICAL EDGE (RANGE)
ORIGINAL IMAGE
SMOOTH+EDGE
MORPHOLOGICAL TEXTURE MEASURES
TEXTURE ENERGY MEASURES (LAWS)
NON-LINEAR IMAGE PROCESSING
LINEAR IMAGE PROCESSING

Figure 11-2 Example Configurations of the Parameterized Image Processing Node

Our approach has been demonstrated on multi-spectral image data sets. Nodes, described in the previous section, are cascaded to form a network on the Firebird Reconfigurable Computer from Annapolis Micro-systems. Parameters are then optimized in hardware with an evolutionary algorithm, for a particular data set and feature of interest. The real-time processing power of the approach was confirmed. The current system has 12 independent input image channels with a 66 million pixels per second throughput on each channel. The total throughput of the system has therefore approximately 752 million pixels per second.

Figure 11-2 illustrates the system output for a golf course feature identification problem. On the far left is a visible channel of the original multi-spectral image used for training. Second from the left is the output image of optimized Reconfigurable Computer on the training image. Second from the right is a visible channel of a second multi-spectral image that as used for testing. On the far right is the Reconfigurable Computer output when it is applied to this test image.

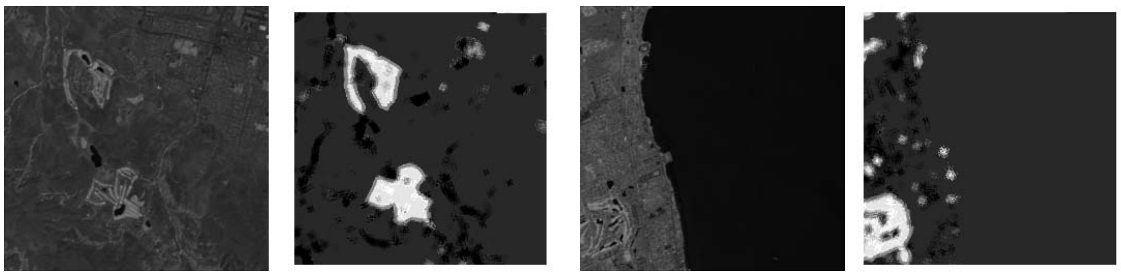


Figure 11-3 System result for the golf course problem.

Extensive testing has shown that the system has comparable accuracy to advanced software solutions. Figure 11-4 compares our system to two advanced software systems known as GENIE and AFREET. GENIE uses a genetic algorithm to find image processing algorithms for solving feature extraction problems, and AFREET is based on a technique known as a Support Vector Machine. In Figure 11-4, we compare the systems with a single measure that combines the detection rate and 1 minus the false alarm rate. A perfect classification (100% detection and 0% false alarm) will result in a score of 1000. Each system is trained on 4 different features: clouds, golf courses, urban areas and roads. For each feature, 4 different scenes were used to train (or optimize) the systems. Once trained, the systems are applied to a different set of images to assess generalization (test images).

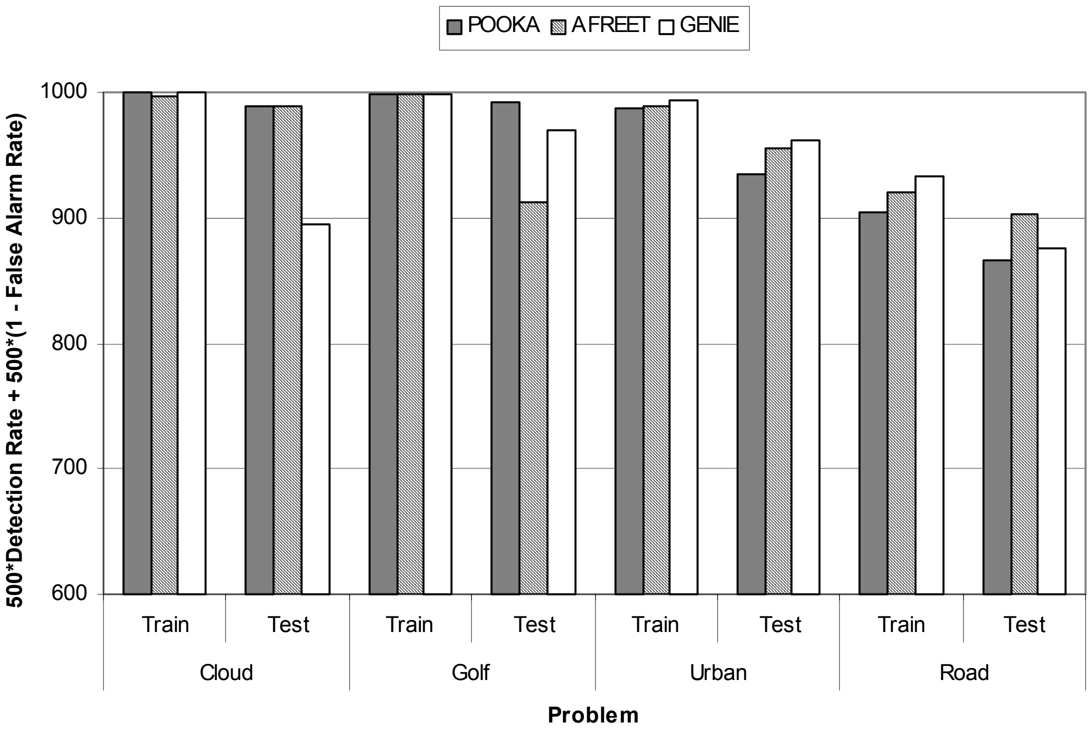


Figure 11-4 Comparing POOKA to advanced software solutions.

For all problems, the training time for the POOKA system was approximately 3.5 minutes. For AFREET, the training time is variable from 9 to 50 minutes, and averaged 19 minutes over all the problems. The GENIE system averaged approximately 19 hours / problem. Once trained, the POOKA system is over two orders of magnitude faster in application to large image databases, or directly at the sensor, than both software systems.

11.3.2 Hyperspectral Infrared Imaging System (HIRIS)

Our multiple approaches to HIRIS signal processing fall into two general categories: Precise Chemical Analysis, and Detection. Existing conventional processing effort has concentrated on precise analysis based on statistical methods, which is a Principal Components Analysis followed by an adaptive matched filter. LANL has selected elements of this processing chain known as HIP as good candidates for acceleration. These elements are matrix multiplications to perform a dot product or a covariance calculation. Alternately LANL has investigated an explicit "physics based" analysis based on classifying and finding spectral end-members followed by spectral fitting. One classification algorithm that is suitable for hardware acceleration is the K-means algorithm and a suitable end-member identification algorithm is the Pixel Purity Index algorithm.

11.3.3 Hyperspectral Image Processing (HIP) Algorithm

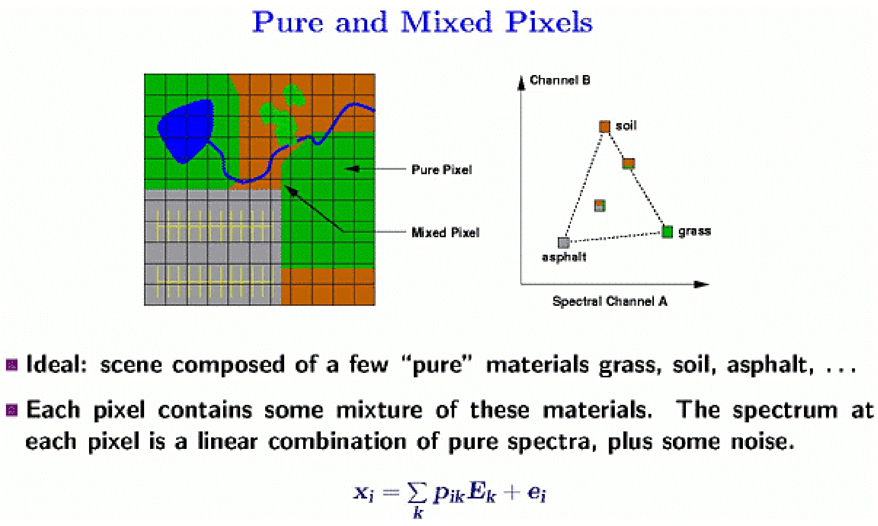
A matched filter processes hyperspectral data by performing a multiply accumulate (MAC) function on every pixel in the image with a filter of the same depth as the pixel. This can be used to give a measure of the amount of a chemical (represented by the filter) that a pixel contains. Typically there are multiple filters in a given application. For example, for 32 filters the FPGA performs 268 million multiplies producing 524 thousand results.

The covariance function is a sum of products calculated by taking the dot product of a spectral plane minus its mean with every other plane in the image minus their respective means. This calculation takes 2.15 billion MACs.

The hyperspectral cube is 128 by 128 by 512 spectral planes of 16 bit data so the image cube size is over 16Mbytes. In order for the system to process data efficiently, the data cube must be stored in memory local to the FPGA and a high speed data path to mass memory is essential. Thus we targeted the Annapolis Microsystems Firebird board with its 64bit/66MHz PCI DMA capability and 36Mbytes of on-board fast SRAM. Since the architecture of the two functions is so similar we are able to switch between functions without a bit-stream reconfiguration by merely reconfiguring registers in the state machine that control data flow. The algorithms were benchmarked on a 733MHz PC. The matched filter took 21 seconds on the PC vs. 1.44 seconds on the Firebird for 32 filters. The covariance acceleration was only a factor of two better due to a internal memory bottleneck in our specific implementation. This could have been corrected but it was not pursued because we believe the other approaches would bear more fruit.

11.3.4 K-means and PPI

The motivation for LANL MDIP work is to be able to perform rapid broad area searches. The dot product is a key step for many of the algorithms being developed for MDIP applications. In the classic application, a single vector is taken as the "matched filter" to be applied to each pixel in the image. LANL has identified two algorithms which both use the dot product. One is referred to as k-means and a second is a computation known as the pixel purity index (PPI). Algorithms based on the dot product, which is implemented as a simple multiply–accumulate lend themselves to reconfigurable computing architectures.



Pure and Mixed Pixels

- Ideal: scene composed of a few "pure" materials grass, soil, asphalt, . . .
- Each pixel contains some mixture of these materials. The spectrum at each pixel is a linear combination of pure spectra, plus some noise.

$$x_i = \sum_k p_{ik} E_k + e_i$$

Figure 11-5 Pixel Purity Index Algorithm

Accelerating Segmentation and Pixel Purity Index (ASAPP) is a project to investigate schemes to speed-up classical algorithms in remote sensing analysis of multispectral imagery. We are working with an algorithm to identify pure pixels (or "end-members") from multispectral data cubes. The purpose of the algorithm is to find a basis set of D-vectors to characterize a D-dimensional image cube. An initial set of "skewers" into the D-dimensional set is used.

The generic PPI algorithm is:

- Set N random D-vectors: skewers
- For each skewer:
    - Compute a dot-product with all the pixels
    - The two pixels having produced the highest and the lowest dot-product are potential pure pixels
- Select the "pure" pixels–i.e. the pixels that have been selected several times as potential pure pixels

A first hardware implementation has been proposed by Dominique Lavenier, visiting scholar from IRISA, France, both for the SLAAC-1 and the AMC Wildforce board. It was based on a pipelined bit-serial approach, and a speed-up of 200 over a software implementation was expected.

A new version has been designed allowing more flexibility and better performance. Thus, the number of channels can be extended from 128 to 224, which is the current data size of the multispectral images. As in the previous version, the implementation targets both the SLAAC-1 and Wildforce board.

The implementation on the Wildforce board for a 224 band image leads to the following computation times with a clock set to 30 MHz:

# SKEWERS	128X128	256X256	512X512
5000	6	23	82
10000	11.5	46	184

(Time in seconds)

Table 11-1 Wildforce Performance

The design implemented on the Wildforce board will fit directly onto the SLAAC-1 board. For a 224 band image, estimation of the computation time is:

# SKEWERS	128X128	256X256	512X512
5000	2	7	28
10000	3.5	14	56

(Time in seconds)

Table 11-2 SLAAC-1 Performance

This estimation is done with a 40 MHz frequency since the technology of this board is more recent. Also, as the SLAAC-1 architecture has wider wire interconnects, we can better exploit resources, and have speedup which is more than twice the Wildforce. This implementation provides a speed-up of 300 compared to a sequential algorithm executed on a 450 MHz PC. In addition to being hand-coded the PPI algorithm was written in Streams-C and debugged in simulation mode on a Linux workstation. It was compiled through the Streams-C hardware compiler and run on the AMC Wildforce.

11.3.5 MDIP Application: K-Means

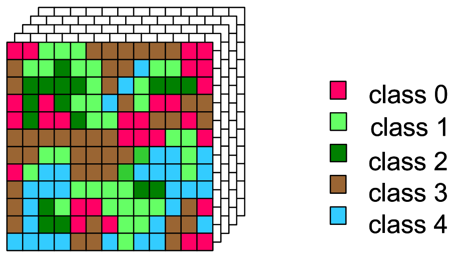


Figure 11-6 K-Means Algorithm Diagram

The K-means algorithm is a traditional approach for identifying and clustering features in images. Like the PPI, it is an intensive computing algorithm, but with high level parallelism opportunities. It is an unsupervised image classification that 'clusters' the pixels into a few numbers of classes such that pixels belonging to the same class have similar spectral properties.

The generic algorithm is:

- **Initialization: Randomly assign pixels to clusters.**
- **Step 1: Find new cluster centers using current cluster assignments.**
    - **Use the mean of pixels assigned to that cluster.**
- **Step 2: Assign pixels to clusters.**
    - **Use minimum distance from pixel to cluster center.**
- **Iterate Steps 1 and 2**

A similar bit-serial approach as the PPI implementation was written in VHDL. The speedup increases as the number of classes (i.e. cluster features) are increased. For a Virtex-1000-based Wildstar, the following speedup estimates were developed (over a 450 MHz Pentium):

Number of Classes	4	8	16	32	64	128	256
DMA speed-up	9	16	39	92	169	253	336

Table 11-3 MDIP Speedup

11.3.6 Hyperspectral Image Processing: Detection

Our approach has been to accelerate the processing chain with a modest reduction in sensitivity. USC/ISI efforts in this challenge, under Dr. Ron Riley, have been focused on algorithmic improvements to the processing chain. To meet the aggressive goal of detection, ISI converted most of the HIRIS processing chain to integer calculations to facilitate implementation on ACS. The covariance matrix is central to the current algorithm for suppressing clutter due to different emission spectrum of various ground material at various temperatures. LANL and USC/ISI postulated that significant image processing could be eliminated by only periodically performing a covariance calculation as part of a principal components analysis since the spectral clutter is expected to change slowly with respect to the frame rate. We knew that jitter in the pointing-angle of the sensor during collection caused significant variations in the covariance, but we postulated that preprocessing the raw data to remove this jitter should lead to improved gas-detection results and would be much faster than computing and inverting the covariance for every image. ISI developed a software model of the hardware processing chain and coded an implementation that used a simple calibration scheme and tested it at LANL on HIRIS data. We found that, although auto calibration of the spectrum improved the gas-plume detection results, it does not replace the requirement to calibrate the spectrum of each pixel in the focal plane array. We believe it is due the fundamental nature of the sensor, which is influenced by airframe vibrations. At this time we believe that each individual frame has to be calibrated independently.

11.4 Results

In a programmatic sense HIRIS has been a disappointment due to factors outside the scope of our ACS effort. There has been a great state of uncertainty in the hyperspectral programs at the national level. The result was instability in the programs we targeted such that the goals and problems to be addressed or even the existence of a program changed faster than we could respond. Despite this we demonstrated our ability to accelerate both statistical and explicit image processing and thus we are well prepared for the needs of the next generation Hyperspectral camera.

The POOKA system was deployed at the National Imagery and Mapping government agency (NIMA) for beta testing since March of this year. The system has been successfully applied to practical broad area search problems and attracted the attention of a second government customer. We expect to deliver a second system in June of this year.

11.5 Related Publications

- Dominique Lavenier, James Theiler, John Szymanski, Maya Gokhale, Janet Frigo, "FPGA Implementation of the Pixel Purity Index Algorithm," SPIE 2000, Nov. 2000
- Jan Frigo, Maya Gokhale, Dominique Lavenier, Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective," FPGA 2001.
- Maya Gokhale, Jan Frigo, Kevin McCabe, James Theiler, Dominique Lavenier, "Early Experience with a Hybrid Processor: K-Means Clustering," ERSA 2001.
- Dominique Lavenier, James Theiler, John Szymanski, Maya Gokhale, Jan Frigo, "FPGA Implementation of the Pixel Purity Index Algorithm," SPIE 2000
- Miriam Leeser, Pavel Belanovic, Michael Estlick, Maya Gokhale, John Syzmanski, James Theiler, "Applying Reconfigurable Hardware to the Analysis of Multispectral and Hyperspectral Imagery," SPIE 2001.
- Reid Porter, Maya Gokhale, Neal Harvey, Simon Perkins, Cody Young, "Evolving Network Architectures with Custom Computers for Multi-Spectral Feature Identification," Third NASA/DoD Workshop on Evolvable Hardware, 2001
- R. Porter, Evolution on FPGAs for Feature Extraction, in Electronic Engineering Ph.D. Thesis, Queensland University of Technology: Brisbane (2001).
- J. Theiler, J. Frigo, M. Gokhale, and J. J. Szymanski. "Co-design of Software and Hardware to Implement Remote Sensing Algorithms.' Proc. SPIE 4480 (2001) 86--99. {Note}: Invited Paper
- R. Porter, M. Gokhale, N. R. Harvey, S.J. Perkins, and C. Young. "Evolving a spatio-spectral network on reconfigurable computing for multispectral feature identification." Proc. SPIE 4480 (2001) 108
- R. Porter, M. Gokhale, N. Harvey, S. Perkins, and C. Young. "Evolving Network Architectures with Custom Computers for Multi-Spectral Feature Identification." In 3rd NASA/DoD Workshop on Evolvable Hardware, Long Beach, California.} (2001).
- D. Lavenier, J. Theiler, M. Gokhale, J. Frigo, and J.J. Szymanski. "FPGA Implementation of the Pixel Purity Algorithm for Hyper-Spectral Images." Proc SPIE 4212 (2000) 30--41.
- J. Theiler, M. Leeser, M. Estlick, and J. J. Szymanski. "Design Issues for Hardware Implementation of an Algorithm for Segmenting Hyperspectral Imagery." Proc. SPIE 4132 (2000) 99--106.
- J. Theiler, D. Lavenier, N.R. Harvey, S.J. Perkins, and J.J. Szymanski. "Using blocks of skewers for faster computation of Pixel Purity Index." Proc. SPIE 4132 (2000) 61--71.

- R. Riley, N. Manukian, “Atmospheric correction of hyperspectral imagery by statistical spectral smoothing," in Image and Signal Processing for Remote Sensing VII, Sebastiano Bruno Serpico, Editor, Proceedings of SPIE Vol. 4541 (2002).

12 FINAL STATUS REPORT ON STREAMS-C

Dr. Maya Gokhale, Dr. Kevin McCabe
Los Alamos National Laboratory

12.1 Introduction

Los Alamos National Laboratory (LANL) Nonproliferation and International Security Division (NIS) provides technology to a wide range of DoE and DoD agencies. NIS supports a research program called Deployable Adaptive Processing Systems (DAPS) that seeks advanced signal-detection and feature-recognition techniques for RF and multi-dimensional image sensor projects such as Aldebaran/ Capella/ Cibola/ Bellatrix, Caliope, MTI, HIRIS/VIPER, SHS, and RULLI, as well as other government-sponsored initiatives. Our role in SLAAC has been to direct and validate DARPA technology via technology insertions into ongoing DoD programs. In 1999 Dr. Maya Gokhale joined LANL and brought her existing ACS funded Streams-C tool development program. The ACS program identified the Key Technological Challenges to reconfigurable computing. LANL supported SLAAC approaches to these challenges in the following ways:

- Building Blocks Inadequate - LANL drove the specification of new configurable computing modules and boards and supported their implementation. LANL developed and described new system-level heterogeneous architectures necessary for our challenge applications.
- Tough to program - Streams-C is a hardware/ software co-design compiler that supports automatic generation and scheduling of hardware and software processes via a streaming data model. It is the first open source C-to-Hardware compiler.
- Minimal runtime support - LANL developed a machine learning based image processing system that used automatic runtime configuration at the datapath level.
- Technology immature/risky - LANL has supported commercial suppliers to DoD by transferring LANL technology to Catalina Research Inc, which resulted in their Chameleon product line. Although Streams-C is not inherently hardware specific, our initial release targets Annapolis Microsystems's Firebird. LANL has developed applications for that board. Both companies have seen an increased customer base as a result.
- Problem formulation - LANL promoted and developed in concert with UCLA and USC/ISI a generic Digital Receiver Library of FPGAs components previously considered to be in the ASIC domain. LANL presented and developed new algorithmic approaches to multi-spectral and hyper-spectral image processing with support from USC/ISI.

12.2 Background

Streams-C is a language that allows users to write a high-level program (using mostly C code) consisting of software and hardware processes, with the intention of running the hardware processes on a field programmable gate array (FPGA), thereby gaining speed on certain applications such as image and signal processing algorithms. The Streams-C project has succeeded in implementing the language, so that the user can first simulate their program, then "synthesize" it, where they run the software processes on the host machine and the hardware processes on the FPGA. The implementation is available on the Red Hat Linux platform, using an Annapolis Micro Systems Firebird FPGA.

12.3 Technical Approach

The concept of stream-based computation is a fundamental formalism for high performance embedded systems, which is characterized by (multiple) streams of data produced at a high rate, with complex operations performed on the incoming data. The Streams-C language supports this computational model with a minimal number of language extensions and library functions callable from a C program. The Streams-C compiler targets a combination of software and hardware, allowing the application developer to annotate a parallel process as being either hardware or software.

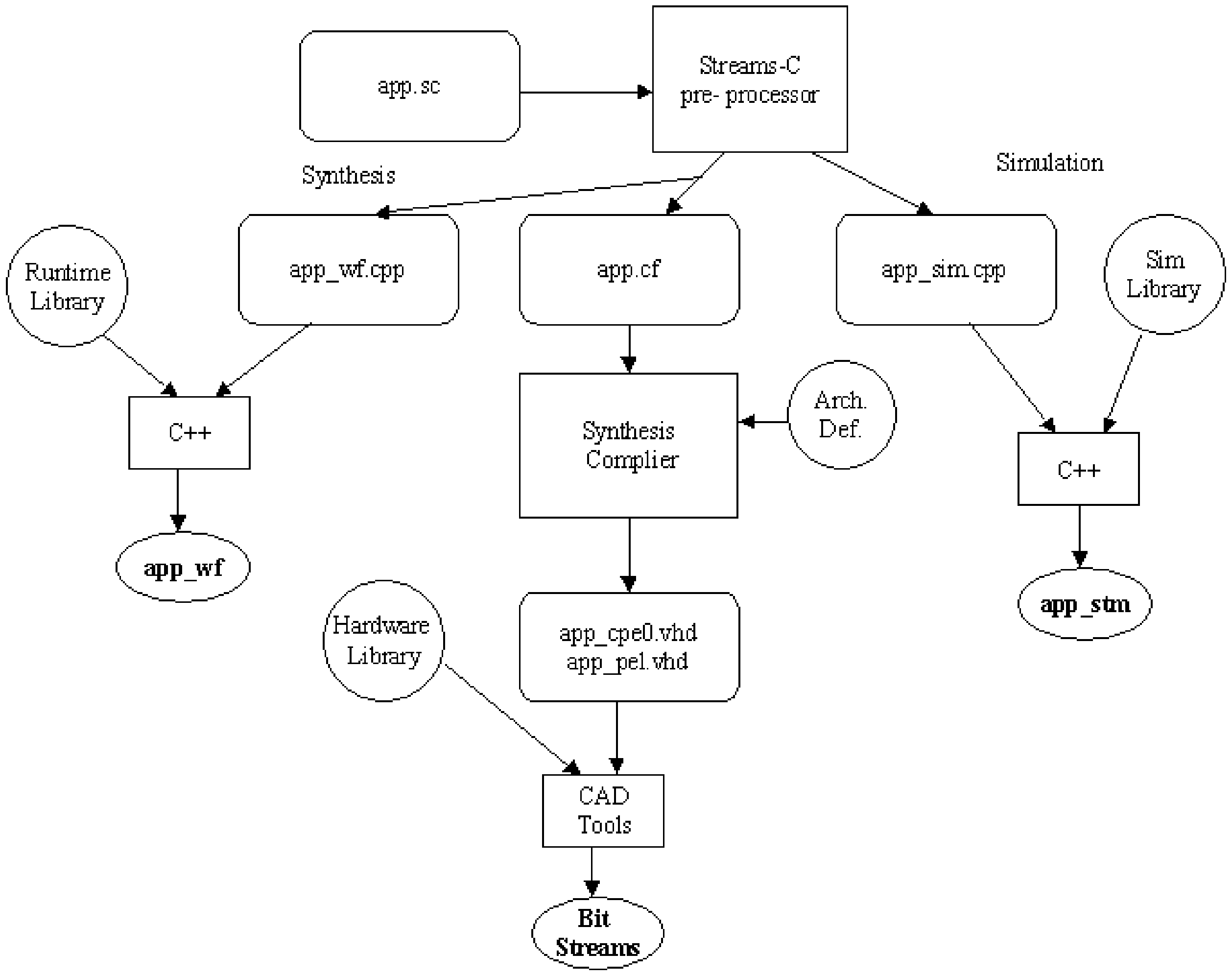


Figure 12-1 Streams-C Development Flow

For computation occurring in hardware, the compiler generates RTL VHDL for a target FPGA board containing one or multiple FPGAs, external memories, and interconnect. For computations occurring in software, the compiler generates C code using threads for parallel software execution. The language extensions allow the programmer to allocate variables to registers on an FPGA and define the variables' bit lengths; assign variables to external or on-chip memories;

define concurrent processes, either in software or hardware; define stream connections between processes; and read/write streams to communicate data between processes. The processes operate asynchronously, and synchronize through stream operations, which may occur anywhere within the body of the process. A distributed memory model is followed, with local state belonging to each process and inter-process communication via streams. The extensions include board-level mapping directives to give the applications developer control over the assignment of processes to hardware components (FPGA chips) and of streams to communication media (interconnection wires) on the target application board.

A software library using POSIX threads provides concurrent processes and stream support in software. Thus the software libraries support a dual function: when all processes are mapped to software, the system provides a functional simulation environment for the parallel program. When processes are mapped to a combination of software and hardware, the software libraries are used for communication among software processes and between software and hardware processes. Hardware libraries are used for communication among hardware processes and for the hardware side of communication to software processes. Figure 12-1 shows the software development flow for applications using the Streams-C compiler.

The C programmer does not have to be concerned with synchronizing state machines, or other hardware timing events occurring within hardware processes. The compiler-generated state machines control sequencing and loops. The compiler can automatically pipeline loops so that no manual pipelining is required. The compiler doesn't just target inner loops – it maps an entire process (which might contain a succession of loops) to hardware. This is beneficial so that the results stay in place for the next loop to take them. Also, multiple independent loops can run concurrently in hardware at the same time. In addition, inner loops are automatically pipelined in the hardware, using a synchronous pipeline with provision for stall conditions due to data unavailability.

12.3.1 C Subset and Extensions

All C syntax conventions are followed for Streams-C. The code appearing in the body of a Streams-C process does not introduce any non-C syntax. New constructs appear in the form of predefined types and intrinsic functions and in directives (in comment blocks) to the sc2 compiler. Streams-C does not accept generalized pointers or dynamic memory allocation.

Extensions include pre-defined types for non-standard bit lengths; intrinsics to insert and extract bit vectors; and intrinsics to send and receive stream communications.

12.3.2 Application Study - original Streams-C compiler

This application study of the Streams-C compiler presents four applications written in Streams-C and compiled to the AMC Wildforce board containing Xilinx 4036 chips. Those same applications have been hand-coded in a combination of RTL and structural VHDL. We compared performance of the generated code with the hand-optimized code and found that the compiler-generated designs are 1.37 - 4 times the area and 1/2 - 1 times the clock frequency of the hand designs. We found that the compiler, based on the SUIF infrastructure, can be greatly improved through various standard compiler optimizations that are not currently being exploited. Thus we developed and are currently releasing a public domain version of Streams-C, sc2, to better optimize and target the Virtex chip.

12.3.3 Retarget to the AMS Firebird Board

Streams-C hardware libraries for the AMS Firebird board, which contains a single Xilinx Virtex-E on a 64-bit PCI bus, were developed for the sc2 release of the compiler. The hardware streams library encapsulates the data flow synchronization between stream reader and writer. The combination of compiler-generated computation nodes with the hardware streams library allows applications developers to target FPGA boards from a high level concurrent language. A unique feature of Streams-C is the architecture definition file. The user can specify the desired FPGA board architecture in this file. The compiler uses the architecture definition to target different board architectures.

We have completed testing the sc2 compiler-generated features such as streams, signals, external memory, and block ram with the VHDL simulator and in hardware. Examples using multiple software and hardware processes, pipelined loops, signals, streams to load and unload external memory and block ram have been completed. We can load and unload external memory and block RAM from the host. More extensive testing of applications such as the K-means clustering algorithm and a poly-phase-filter bank are on going (see below).

A design tools study completed this past year, provides a comparative analysis of how the sc2 compiler performed compared to similar commercially available tools. All designs were targeted to a Xilinx XCV1000-6-CG560 for the purposes of this study.

The polyphase filter using two filter banks of size 8 and 16 are implemented on one chip. The input data comes onto the FPGA via a stream from the host and is unsigned, fixed point, 8 bit data. The coefficients are unsigned, fixed-point 12 bit values. The Streams-C version resulted in 18% area utilization with a speed of 58.5MHz. The design delivers a result every clock cycle. Note that the C code was automatically pipelined when the pragma SC_pipeline was inserted in the while loop. Streams-C design-to-implementation took one day.

Two of the four passes in the FFT algorithm were manually unrolled in Streams-C. A previous proprietary version of Streams-C performed loop unrolling with a pragma directive. This feature has subsequently been added to the public domain version of the compiler. (The entire algorithm written in nested loop format did not place and route). Better performance was obtained by manual resource sharing. Two versions of the FFT were generated – one pipelined with the use of a compiler pragma to pipeline - and the second one non-pipelined. Results for both the unpipelined case 64% area utilization at 66MHz and the pipelined case 37% area utilization at 78MHz were comparable to the commercial tools such as Adelante Builder. Streams-C design-to-implementation took a few days.

12.3.4 Design Environment

Streams-C is available now in source form for non-commercial purposes. It will shortly be available in source form on the LANL web site rcc.lanl.gov.

The Streams-C tool runs on Linux with gcc. In these experiments we used Red Hat Linux 7.2 and the GNU gcc 2.96 C^{++} compiler. The C^{++} simulation code generated by Streams-C runs on Linux. A third party synthesis tool synthesizes the RTL VHDL output.

Streams-C is built into the Stanford University Intermediate Format (SUIF) national compiler infrastructure (suif.stanford.edu), version 1.3. SUIF must be installed in order to run the Streams-C compiler.

One of the primary advantages of Streams-C from the hardware/software developers standpoint is that a software only functional simulation, can be compared against the hardware. This is a feature built into the tool. A testbench, the mti_vcom.do file is provided, for ModelSim, the hardware simulator, which enables direct evaluation of the hardware generated.

The software simulation libraries were written in C^{++} and use a threads package as well, one thread per Streams-C process. The simulation facility enables the user to run their program in software only, so they can check for program correctness. Most of the process communication and synchronization issues can be resolved in simulation.

The synthesis compiler implements the heart of the Streams-C language; it converts the user's Streams-C code for the hardware processes into the VHDL code that can run on the FPGA. The compiler uses the SUIF compiler to do the initial processing of the code. A scheduler and code generator were written for the latter compilation stages.

The runtime libraries enable the software processes to execute and communicate with hardware processes running on the FPGA. The libraries communicate with the FPGA via a protocol of hardware register reads and writes for stream and signal communication. Memory and block RAM reads and writes are implemented with the library calls provided for the Firebird FPGA board.

The VHDL library includes the VHDL components (code) needed for communication between the hardware processes and between hardware and software processes. Specifically, these components implement the stream, signal and memory mechanisms for the FPGA.

12.4 Results

Streams-C is the first public domain C-to-hardware Compiler. It is a true subset of C using a familiar message passing model. Approval was obtained to make all the Streams-C code "open source", so version 1.0 Beta was sent upon request to some sites in early April 2002. The sites that now have the Streams-C release include AFRL Eglin, University of Virginia, University of Cincinnati, New Jersey Institute of Technology and University of New South Wales. The 1.0 Beta release includes the simulation libraries, the synthesis compiler, the runtime libraries and the VHDL libraries. The release also contains the Streams-C reference manual and documentation on the simulation and runtime libraries. The example applications included in the release are two stream applications and two memory applications. Future releases will also contain a GUI, which helps with writing the Streams-C programs, a hardware manual, and more applications such as K-means, Pixel Purity Index and FIR Filter.

Testing has been done on Streams-C applications, which communicate via streams, signals, memory and block RAM. Further stress testing will be done to test larger numbers of simultaneous connections between software and hardware processes. Streams-C was validated using realistic benchmarks based on LANL applications in our Challenge areas. We were thus able to compare development time and performance metrics for the following four signal processing algorithms.

- RF data processing
    - **Poly Phase Filter Design**: 4 tap FIR filter.
- Image processing
    - **Contrast Enhancement**: histogram of image, grayscale creation (table), and image remapping.
    - **Pixel Purity Index Algorithm**: 2D array dot products.
    - **K-means Clustering Algorithm**: subtractor/adder and memory module.

12.4.1 Specific Performance Results

Streams-C: area increased 2-4X over hand coded VHDL while clock frequency decreased by 1 - 1/2X. Productivity increased 10X yet Streams-C PPI on an older technology Xilinx 4036 FPGA is still 12X faster than a Pentium450. Complete documentation of detailed technical results in many cases can be found in the published literature referenced below.

12.5 Related Publications

- Maya Gokhale, Janice M. Stone, Janette Frigo, "Parallel C Programming of Reconfigurable Computers: the Streams-C Approach," presentation at the HPEC2000 conference
- Dominique Lavenier, James Theiler, John Szymanski, Maya Gokhale, Janet Frigo, "FPGA Implementation of the Pixel Purity Index Algorithm," SPIE 2000, Nov. 2000
- Jan Frigo, Maya Gokhale, Dominique Lavenier, "Evaluation of the Streams C-C to FPGA Compiler: An Applications Perspective," in Proceedings of FPGA 2001.
- Maya Gokhale, Jan Frigo, Kevin McCabe, James Theiler, Dominique Lavenier, "Early Experience with a Hybrid Processor: K-Means Clustering," ERSA 2001.
- J. Theiler, J. Frigo, M. Gokhale, and J. J. Szymanski. "Co-design of Software and Hardware to Implement Remote Sensing Algorithms.'' Proc. SPIE 4480 (2001) 86--99. {Note}: Invited Paper

13 FINAL STATUS REPORT ON THE IR/ATR CHALLENGE

Dr. John Villasenor
University of California, Los Angeles

13.1 Introduction

In June 1997 the University of California Los Angeles was contracted through the University of Southern California within the System Level Application of Adaptive Computing (SLAAC) project to explore the use of reconfigurable field-programmable gate arrays (FPGAs). In particular, an implementation of an automatic target recognition (ATR) system on a board utilizing that technology was the main goal of the project. There were two different algorithms for ATR: the Sandia UCLA design and the Night Vision Laboratory (NVL) design. UCLA did most of its work on the NVL-ATR algorithm. Generally, UCLA was to examine whether significant performance gains or chip space gains could be achieved using reconfigurable FPGAs. If these gains could be achieved, then UCLA was to implement the ATR algorithm in hardware.

The SLAAC board used had five FPGAs on it. One was used for static, application-specific system architecture (for example, memory interfaces and I/O routing). The other four were used for adaptive computing. Eventually the boards would be installed on a tank, although UCLA did not participate in this part of the project. UCLA, while not charged with designing the target recognition algorithm, did make minor modifications to optimize performance. UCLA did extensive testing and debugging in both software simulations and on actual hardware. For a detailed description of this part of UCLA's work, see the section below entitled "NVL-ATR Description".

In addition to the ATR system, UCLA also worked with Los Alamos National Laboratory to design an adaptive, scaleable polyphase filter bank. Later, UCLA also designed an adaptive time-frequency windowing (Gabor) filter. This topic is covered in more detail below under the section titled "Gabor Filter Bank". A general description of what UCLA was expected to perform follows. A more detailed timeline can be found in a later section of this report entitled "Timeline of UCLA's Work".

13.1.1 <u>Year 1</u>

UCLA was to develop a third generation ACS computing module targeted toward solving the problem of scaleable ATR.

13.1.2 <u>Year 2</u>

UCLA was to design CAD tools to assist in the design and testing of the NVL-ATR project. These tools were to be able to support multiple rotations, and they were also to be used across different FPGAs with different sizes.

13.1.3 <u>Year 3</u>

A detailed survey of available adaptive computing systems, including those from both commercial sources and those within the DARPA research community, was to be performed.

Also, UCLA was to design a first generation implementation of the Round 0 processing for the NVL algorithm (an explanation of this can be found below). This year UCLA also began work on the polyphase filter bank.

13.1.4 Year 4

The implementation of the NVL-ATR algorithm was to be completed and shown to work on the selected platform. Extensive testing was to be done, both before and after the VHDL was supplied to ISI. Also, the polyphase filter bank was finished and the time-frequency windowing (Gabor) filter was started.

In December 1999, UCLA finished work on the NVL-ATR project and submitted the working VHDL to ISI. The UCLA team continued to support the project and assist with further testing and debugging after the submission. By early 2001, the polyphase filter was completed. The Gabor filter's VHDL was finished in September 2001.

13.2 IR-ATR Algorithm

The IR-ATR (Infrared Automatic Target Recognition) algorithm was developed at Night Vision Laboratories (NVL) under the ATR Relational Template Matching program (ARTM) to enable a real-time image cueing system for M1 tank operators driving at night. The system operated as follows: the tank operator, viewing a monitor inside the cockpit, would see an infrared image of the terrain ahead with probable targets clearly marked. The system would not be capable of categorizing targets as friend or foe, (the tank operator must still decide which targets pose a threat) but the task was simpler when the position of the targets was already known.

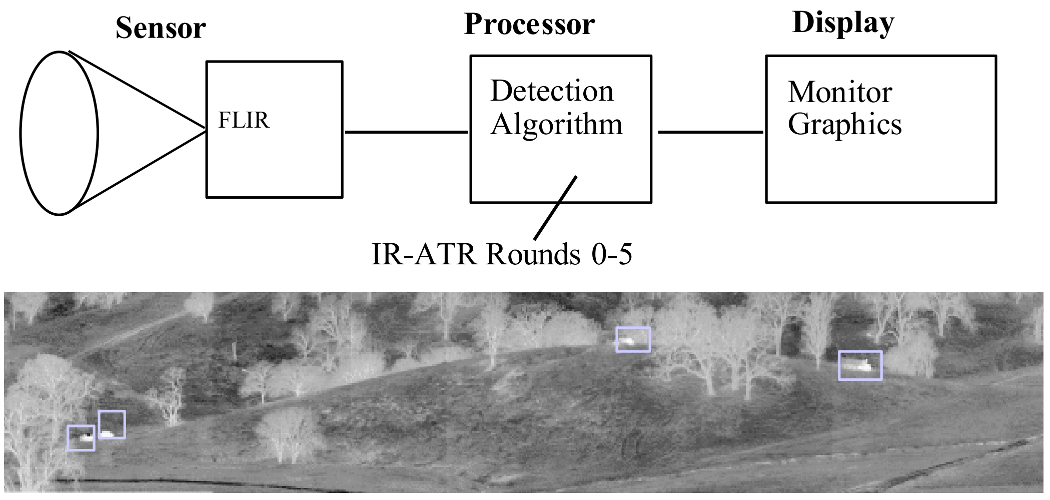


Figure 13-1 Image Cueing System block Diagram w/ Result Image

To implement the real system, operating constraints such as speed and form factor were tantamount. The system needed to be fast enough to process large, 180° or more views of high-resolution image data, with refresh rates of no less than 10 Hz. The system also had to be small enough to fit into the tight space of a tank, in this case, a rugged equipment enclosure with only 2 VME slots available. The IR-ATR algorithm was basically the mathematical description of how to process the image and find the targets in it. The algorithm had the form of a 6-level decision tree, with the top level (Round 0) representing the largest computation bottleneck. The first computation partitioning decision in this project was to allocate one VME slot to the computation of Round 0, and the other VME slot to the computation of Round 1-5.

13.2.1 Round 0

Round 0, the first step of the NVL IR-ATR algorithm, was a correlation calculation based on a background and a target template. The Round 0 calculation was performed at every pixel location of an input image to find target-like shapes (i.e. pixel locations having a high correlation value). This small percentage of pixel locations advanced to Round 1 of the algorithm for further processing using a different type of calculation. All subsequent rounds, 6 in total, followed this type of narrowing decision tree, making Round 0 the most processor-intensive step of the algorithm based on the sheer number of pixels that required processing. The Round 0 algorithm was as follows [2]:

```
-Pick 30 B (background) and 30 T (target) pixels according to templates
-Compute mean (M) of B -For all T and B > M
Sh = {(sum of T-M) – (sum of B-M)}/{(sum of T-M) + (sum of B-M)}
-For all T and B < M
Sc = {(sum of M-T) – (sum of M-B)}/{(sum of M-T) + (sum of M-B)}
-Compute R = max(0, $Sh^+ + Sc^+$); Threshold R > 0.65 for target-like pixels
```

13.2.2 Target Templates

The effectiveness of the Round 0 algorithm was heavily dependent on the choice of template; that is, the location of the 30 background and 30 target pixels. These locations were chosen *a priori* according to the morphology of a master form, which was the union of several binary images of military vehicles or targets. Shown below is picture of the Round 0 master form (at top), which was the Boolean union of all the vehicle forms underneath it in the hierarchy.

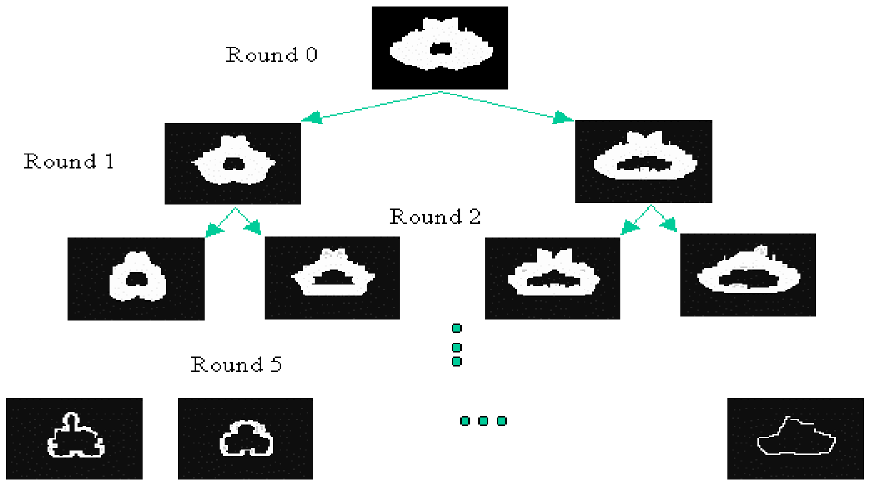


Figure 13-2 IR-ATR Template Hierarchy

The 30 pairs of pixels for Round 0, also sometimes called probes, were selected based on the criteria that they straddle the white, or edge, regions of the master form. This meant that all 30 target probes, located inside the closed contour of the white edge on the master form, had to lie inside the closed contour of all forms underneath the master form in the hierarchy. Likewise, all 30 background probes, located outside the closed contour of the white edge on the master form, had to lie outside the closed contour of all forms underneath the master form in the hierarchy. In this way, the probes' locations, derived from the master form, were appropriate for testing the entire hierarchy of forms as a whole, as the probe set met the criteria for straddling the edge regions of every form in that hierarchy.

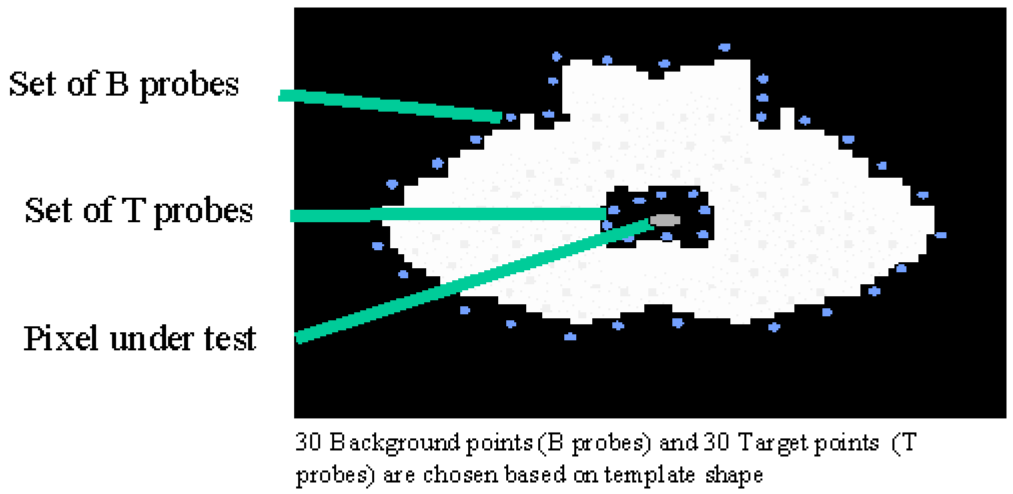

Set of B probes
Set of T probes
Pixel under test
30 Background points (B probes) and 30 Target points (T probes) are chosen based on template shape

Figure 13-3 Round 0 Template

The IR-ATR algorithm was basically a decision tree, with each subsequent round deciding the best matching form from a different row of the hierarchy. While the algorithms of each round differed considerably, each involved a set of relational probes derived from a form similar to the set used for Round 0. Since every parent form in the tree was the union of its child forms, the best matching form in a round narrows the decision space to that particular subtree. The subtree for Round 0 was the entire tree of forms, whereas the subtree left after the final Round 5 was composed of only one form.

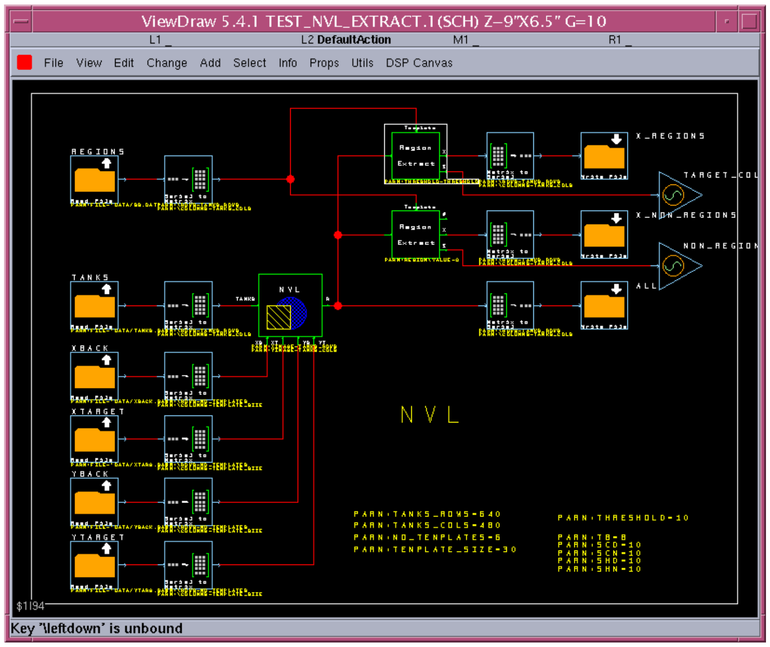
Because the background probes had to lie outside of the master form contour, which was also the largest contour of the hierarchy, the Round 0 templates were much larger than those used in the Sandia UCLA ATR system. In the worst case, a configuration of 60 probes could occupy a rectangle as large as 100 pixels by 60 pixels. Compared with the original Sandia UCLA ATR system, which used templates of 8 X 8 pixels, or even the second generation Sandia UCLA ATR system, with templates of 32 X 32 pixels, this was a very significant increase in size, and also had significant implications for the hardware. The Sandia UCLA ATR system worked by feeding the image pixels serially into a chain of shift registers that formed a 2-D "pixel grid" with the same dimensions as a single template. The computational elements (in the ATR case, the adder tree) were hard-wired to this pixel grid in a pattern dictated by the template; that is, only those pixel locations that were "on" pixels in the template were wired to the adder tree; all others pixel locations stored in the pixel grid were efficiently ignored. The novelty of the Sandia

UCLA ATR system was that the adder trees could be changed on the fly via reconfiguration to accommodate different templates.

Unfortunately, the Sandia UCLA ATR line-delay model was poorly suited to Round 0. If the same system were built for Round 0, not only would it have required that the pixel grid be as large as 100 pixels by 60 pixels, but in addition the pixel depth would have to be increased from 5 bits (for Sandia UCLA ATR) to 12 bits, meaning that each shift register in the pixel grid would have had to more than double in size. The total number of D-flip flops in a Round 0 pixel grid was **12 times** the total number of D-flip flops in the Sandia UCLA ATR pixel grid. However, the availability of configurable logic blocks for Round 0 was the same as for Sandia UCLA ATR (the SLAAC-2 uses the same Xilinx 4085XLA part as the Sandia UCLA ATR system). This much extra hardware simply did not fit on the FPGA.

13.2.3 <u>Precision Analysis</u>

The effects of varying the bit precision of the Round 0 algorithm were investigated using Angeles Design Systems' DSP Canvas tool. DSP Canvas was able to optimize the bit lengths for intermediate values in a calculation with the goal of reducing the amount of hardware used. The tool was very useful for analyzing the Sandia UCLA ATR application; it helped show that reducing the pixel depth from 8 bits to 5 bits did significantly affect the correlation peak location. This finding not only saved hardware (the width of the shift registers in the pixel grid), but more importantly saved clock cycles, as the computation cycle time was directly proportional to the number of bitplanes in the image. The simulation was undertaken in the hopes that a similar savings could be obtained for the UCLA Round 0 implementation.



PARM:TANKS_ROWS=640
PARM:TANKS_COLS=480
PARM:NO_TEMPLATES=6
PARM:TEMPLATE_SIZE=30

PARM:THRESHOLD=10

PARM:TB=8
PARM:SCD=10
PARM:SCN=10
PARM:SHD=10
PARM:SHN=10

Key '\leftdown' is unbound

Figure 13-4 Screenshot of DSP Canvas Simulation

The DSP Canvas simulation ran an optimization script based on a cost function which was defined as: 2^(Sh num bits + Sh denom bits) + 2^(Sc num bits + Sc denom bits) + (16 * image bits). The cost function was heavily weighted by the Sh and Sc bit lengths because the size of the divider was an important concern. It was hoped that if these bit lengths could be reduced sufficiently, a lookup table could replace an actual divider component. The other factor included in the cost function was the image pixel bit length; which it was thought could help reduce the number of clock cycles as in the Sandia UCLA ATR system. The results of a 1000 iteration simulation on a sample image showed that the image bits could have been reduced to 6 bits, the Sh and Sc reduced to 5-7 bits, and the resulting average error would have been approximately 4% deviation from the full precision calculation. While these results indicated that there was copious room for optimization, the image bit reduction was not enough: the pixel grid was still 7 times the size of the Sandia UCLA ATR pixel grid and still too large to fit on the FPGA.

Once the decision was made to abandon the pixel grid/line delay concept for Round 0, it was possible to build a system that used full precision and was still capable of meeting the specified timing constraint.

13.2.4 Hardware Platform: SLAAC-1 & SLAAC-2 Boards

The Systems-Level Applications of Adaptive Computing (SLAAC)[3] project was to design and implement a scalable, distributed platform for adaptive computing systems. The SLAAC-2 [4] board was the most recent product of this design effort, and was built upon an existing architecture known as the SLAAC-1. The SLAAC-1 architecture had 3 user configurable FPGAs (one Xilinx XC4085XLA called the X0 and two Xilinx XC40150XVs or XC4085XLAs called the X1 and X2), and one interface FPGA (called the IF). The X1 and X2 both had four dedicated 256K X 18 bit ZBT SRAMs, while the X0 had two 256K X 18 bit ZBT SRAMs. The X1 and X2 connected to a 72-pin bus beginning at the X0, and all three chips were linked to a 72-pin bus ring. The SLAAC-1 PCI board was used to do preliminary hardware testing of the UCLA Round 0 design. The SLAAC-2 architecture was equivalent to two SLAAC-1 nodes; there were a total of 6 user programmable FPGAs available.

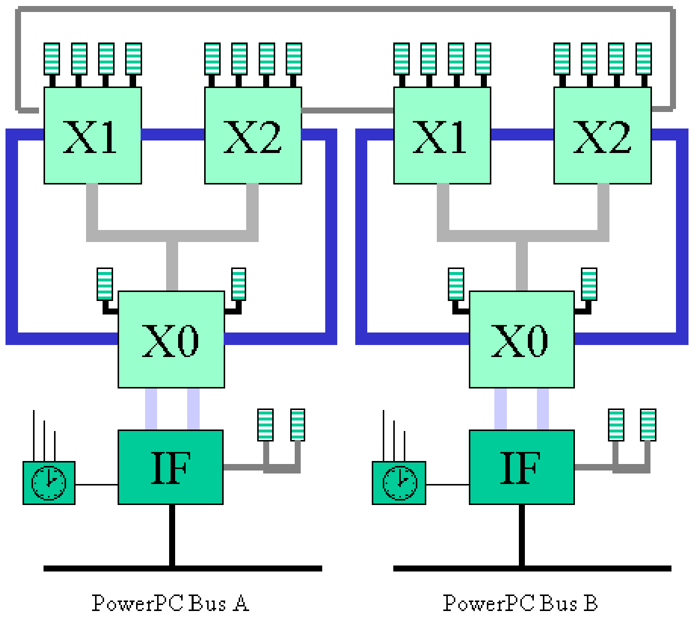


Figure 13-5 SLAAC-2 Architecture

The SLAAC-2 was a mezzanine board designed to be plugged into a modified 6U VME CSPI 2641/A dual PowerPC muticomputer baseboard. It was the SLAAC-2 architecture that was targeted for this Round 0 implementation.

13.2.5 High-Level Partitioning

At the highest level, the IR-ATR algorithm was partitioned across 2 6U VME boards, one of which performed Round 0 calculations (the SLAAC-2), and the other, which performed Rounds 1-5. One level below, the work of the Round 0 algorithm was partitioned across multiple FPGAs. This partitioning decision of how to allocate the FPGA resources was made based on properties of the infrared input images.

Recall that since this algorithm was used for tank combat, the IR-ATR images are generated from a horizontal vantage point, unlike the Sandia UCLA ATR images that were generated from a bird's eye view. For this reason, the objects at the bottom of the image were much closer (and larger) than the objects at the top of the image. The image was accordingly sliced horizontally into 5 depth regions or strips, and each was assigned an appropriate scaling factor that adjusted the normalized template to the proper size. UCLA modified the IR-ATR algorithm with the approval of Night Vision Laboratories to use 4 strips rather than 5.

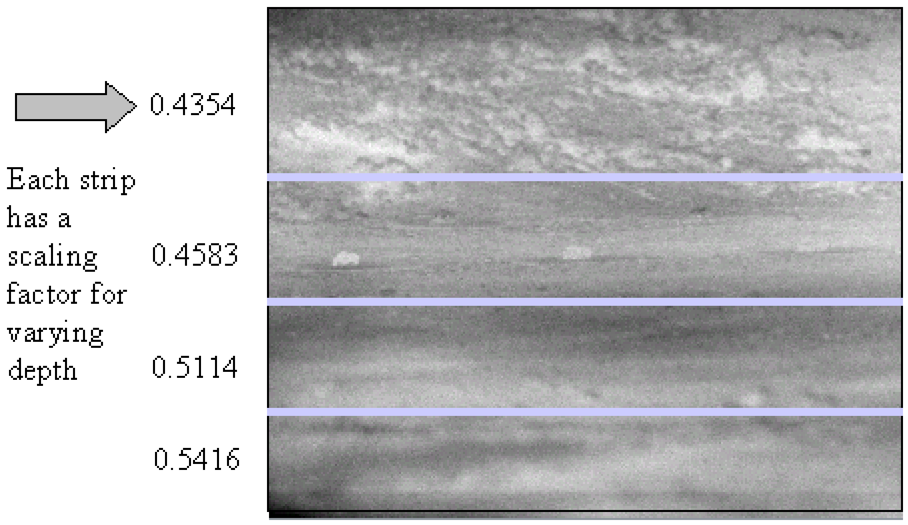


Figure 13-6 Image Partitioning

One FPGA was assigned to process each strip. Recall that the SLAAC-2 had 6 user programmable FPGAs, so this left 2 FPGAs which were used as controllers. This partitioning choice had a significant advantage: since each FPGA only processed one strip, the template scaling factor for that FPGA never changed. If this were not the case, the template size would either have had to be recalculated in the middle of processing, or multiple templates would have had to be stored. This was the case because the template information was stored in read-only memory due to speed considerations.

Incidentally, the strips were not the same size; strip 1 was actually almost twice as large as the other strips. Thus, despite the opportunity to parallelize the processing of the strips, the bottleneck of strip 1 caused an unavoidable lull in processing activity in the other FPGAs as they waited for strip 1 to finish. Despite this apparent inefficiency, the performance of the system was still sufficient to meet the timing requirements for the application.

13.2.6 Memory Usage

Each of the four computation FPGAs had four dedicated 256K X 18 bit ZBT (Zero-Bus Turnaround) SRAMs, while each of the two controller FPGAs had two 256K X 18 bit ZBT SRAMs. The Round 0 implementation, as with many image processing algorithms, was limited mostly by the bandwidth of the input data (that is, the image pixels). A good use of the memory helped to maximize the bandwidth of the input data. In this case, each SRAM contributed one pixel to the computation FPGA each clock cycle, making it possible to meet the timing requirement. Since there were 30 background and 30 target pixels to be examined per image offset, and 4 pixels arrived each clock cycle, the lower bound was 15 clock cycles to finish one step of the algorithm. In actuality, however, it took 20 clock cycles due to pipeline latency. In theory, if there were 60 independent memory channels, the algorithm would have finished in one clock cycle. Likewise, with only 1 memory channel, 60 clock cycles would have been required.

UCLA could obtain the best memory utilization during processing by writing identical copies of the image strips to each of the four SRAMs ahead of time, and then extracting the pixels, as they were needed. Because strip 1 was too large to fit in the SRAM, the top 30 rows of that strip were excluded. Fortunately, leaving out these rows has no effect at all on processing results because these rows were not in the viewable area defined for the correlator.

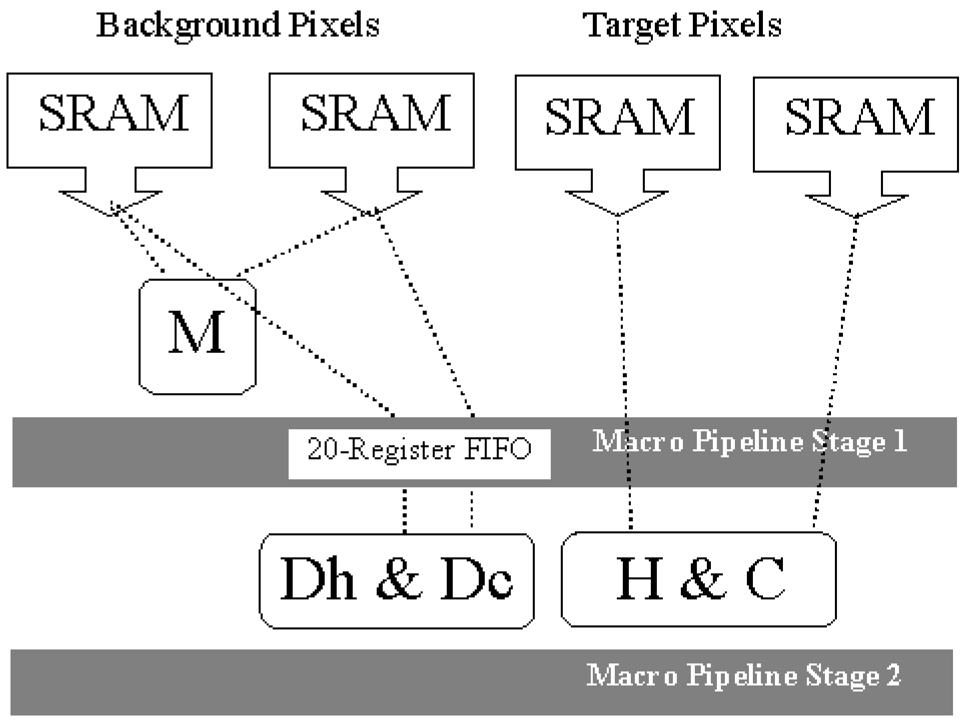


Figure 13-7 Using SRAMs Efficiently

13.2.7 VHDL Design Architecture

13.2.7.1 Computation Unit – X1

The Round 0 computational unit, the X1 FPGA, used a general purpose datapath in the sense that there was no template-specific information contained in the topology or any other characteristic of the datapath itself. Parallelism of the calculations based on obvious dependency relationships yielded the following diagram, which represented the organization of the datapath:

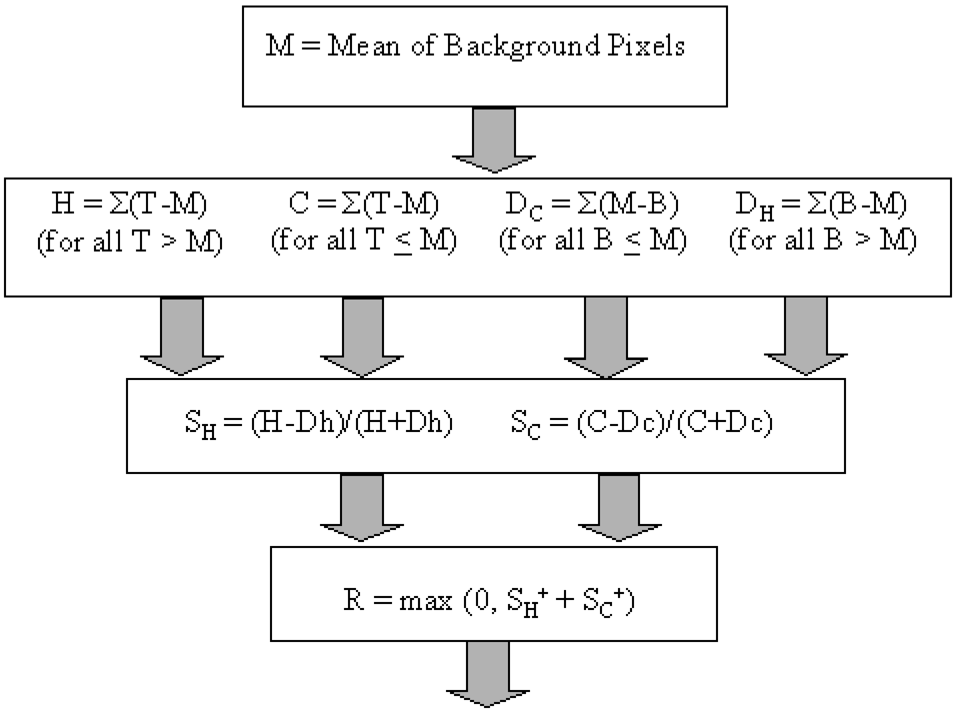

M = Mean of Background Pixels
$H = \Sigma(T-M)$ (for all $T > M$)
$C = \Sigma(T-M)$ (for all $T \leq M$)
$D_C = \Sigma(M-B)$ (for all $B \leq M$)
$D_H = \Sigma(B-M)$ (for all $B > M$)
$S_H = (H-Dh)/(H+Dh)$
$S_C = (C-Dc)/(C+Dc)$
$R = \max(0, S_H^+ + S_C^+)$

Figure 13-8 Parallelizing Operations

The design of all arithmetic components in the datapath was originally done using structural VHDL. However, it became clear once a large portion of the datapath was complete that despite the good design style, the automatic place and route tools would not yield the necessary speed. At this point, the VHDL arithmetic components were replaced by much faster relationally placed macros (RPMs). The macros for XC4000 series FPGAs that were available through the Xilinx Foundation Series 1.5 tools (Logiblox and Xilinx Core Generator) provided a tremendous improvement in speed not only for the datapath, but also for the place and route operation: from up to 23 hours to about 2 hours. This was because the macros were placed as single contiguous units, rather than in a random placement of CLBs. The macros currently used by the design include comparators, adders, multiplexors, accumulators, ROMs, registers, inverters, clock dividers and counters.

The datapath relied heavily on pipelining to decrease critical path and thereby increase speed. The main datapath was divided by 4 pipeline stages, advanced by a macroclock 20 times the period of the system clock. Between these pipeline stages were up to 4 regular pipeline stages advanced by the system clock (more stages were used in certain places to increase speed). This pipeline configuration evolved naturally from the separate stages of the algorithm.

Three places in the algorithm required a division: the Hot and Cold correlation values (which were added together to obtain the final correlation value), and the calculation of the mean of background pixels. The Hot and Cold correlation values each used a 16-bit Sweeny-Robertson-Tocher (SRT) radix-2 divider, written in structural VHDL. These divider components contained the critical path of the Round 0 design because no macro was available to replace their function. Although the DSP Canvas simulation results indicated that a lower bit precision would suffice, which could reduce the critical path, 16-bits were used because the critical path did not prevent UCLA from meeting the timing requirement. However, automatic place and route of a 17-bit SRT divider for the mean of 30 background pixels created a prohibitive critical path, so the calculation was instead computed by a simple approximation that makes use of dividing by a power of 2.

$$Mean = \overline{B} = \frac{1}{30}\left[\sum_{i=1}^{30} B_i\right] \approx \frac{1}{32}\left[B_1 + B_2 + \sum_{i=1}^{30} B_i\right]$$

An effective way to counteract the imprecision introduced by this mean approximation was to delay the integer divide by 32 until later in the algorithm.

While the datapaths for all four computation FPGAs were the same, the address generators were different for each FPGA. All template-specific information was concentrated in the address generators, specifically in four 15-element ROMs (one used for each SRAM). The ROMs contained offsets that were added to the address of the current pixel under test to extract the 30 background and 30 target probes corresponding to that pixel. Image pixels at the probe locations arrived from SRAM at a rate of four per clock cycle, and were delivered directly to the computation datapath. Because each FPGA used a differently scaled version of the main template, the contents of the ROMs varied across FPGAs. In addition, the sizes of the strips were not the same, so counters in the address generator differed across FPGAs.

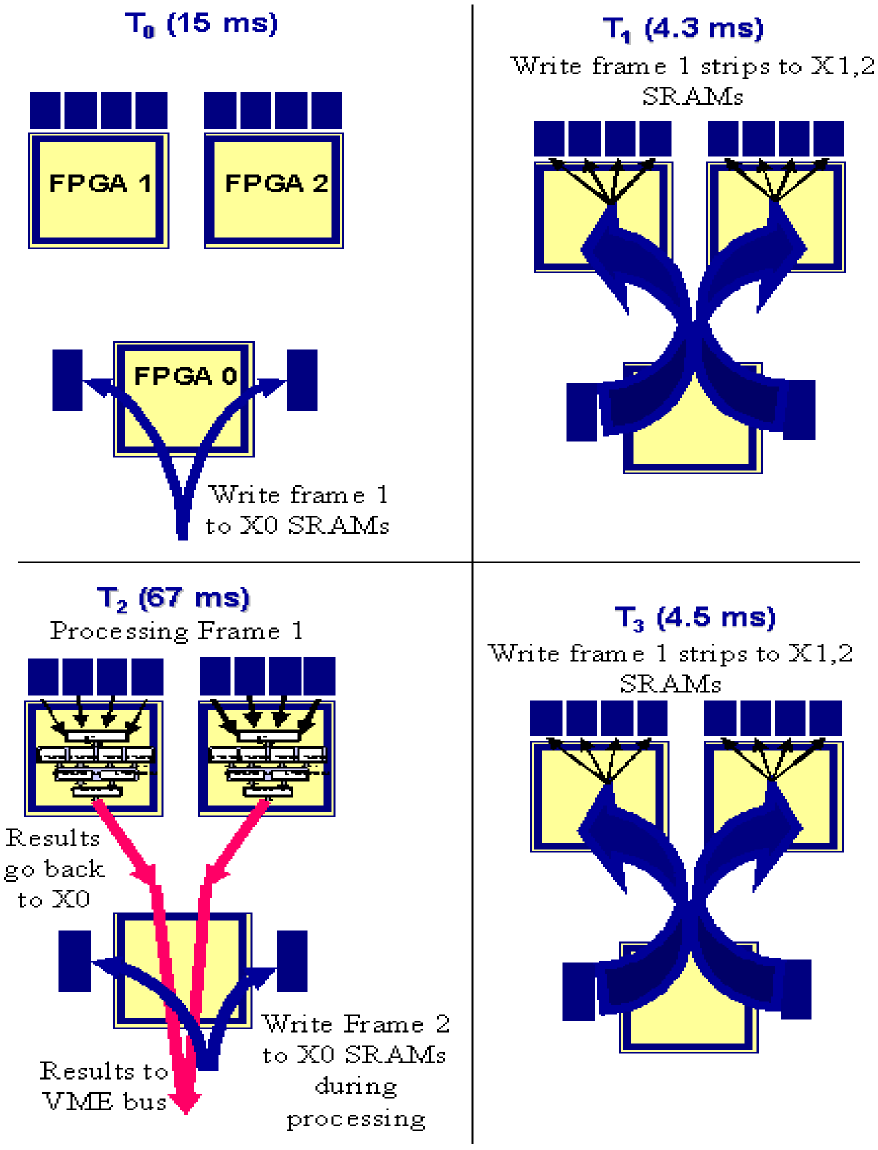
Control was centered in a Moore machine located in the address generator. The VHDL for it was generated automatically using a graphical state machine editor in the Xilinx Foundation Series 1.5 package.

13.2.7.2 Controller – X0

The controller FPGA, the X0, acted as master to the X1 and X2 FPGAs. The X0 acquired the image frame from the FIFO, segmented the frame into 2 strips, cached the strips in memory, and eventually sent one image strip to each computation FPGA. The X0 then initiated processing, gathered results as they arrived, and wrote the results to the output FIFO. The design for the X0 logic also employed macros and several pipeline stages; control was also centered in a Moore machine. The X0 changed state based on two inputs: counters which tallied the number of pixels that arrived at the input FIFO, indicating that a frame was complete, and processing completion signals sent by the computation FPGAs.

13.2.8 Operation

The operation of the Round 0 system is illustrated by Figure 13-9 on the following page. The figure shows the sequence of events after powerup, starting with the initial latency where the first image frame was cached by the X0, taking time T0. After the image frame (actually just 2 of the 4 strips of the image, as this was only one of the two SLAAC nodes) was cached, it was sent to the X1 and X2, taking time T1. Now processing began, and continued uninterrupted while the second image frame was cached by the X0. This whole step took time T2. Because caching of the second image frame was complete before processing of the first frame, caching the second image frame did not add overhead to time T2. When processing was complete on the first image frame, the second image frame was sent to the X1 and X2, taking time T3, which was equal to time T1. The cycle continued. So, the complete cycle for one image frame consisted only of the time to send the cached image to the X1 and X2, and processing time for the largest strip of the image, or T1 + T2. Shown on the next page is a visual timeline of the steps of operation, with a hypothetical 66 MHz clock.



T0 (15 ms)
FPGA 1
FPGA 2
FPGA 0
Write frame 1 to X0 SRAMs
T1 (4.3 ms)
Write frame 1 strips to X1,2 SRAMs
T2 (67 ms)
Processing Frame 1
Results go back to X0
Results to VME bus
Write Frame 2 to X0 SRAMs during processing
T3 (4.5 ms)
Write frame 1 strips to X1,2 SRAMs

Figure 13-9 Operation on one SLAAC-1 node (cycle time is T1 + T2)

13.2.8.1 Reconfiguration

The UCLA Round 0 implementation did not at this point make use of FPGA run-time reconfiguration. Because of the rigid timing constraint that the cycle time be less than 100ms, there was not enough time to perform a full-chip reconfiguration without skipping frames (for Xilinx XC4085XLA, configuration times are still approximately 200ms). However, it would have been desirable to be able to change the template during processing, and this could have been accomplished in two different ways. One way would be to use partial reconfiguration. The Xilinx Virtex series of FPGAs were reportedly capable of configuring just a single column of CLB blocks on the order of 10ms. Since the only template-specific information was contained in the four address generator ROMs, these ROMs could be hand-placed in a single column of CLBs (or a few columns), and could in this way be reconfigured very quickly. Another way that was far simpler and faster was to change the ROMs to on-chip RAMs, and change their contents directly without reconfiguration. Since there were only 60 probe locations stored on each FPGA, with a 60 MHz clock, it would have taken 1μs to change templates. Unfortunately, the on-chip RAM macros tested were not nearly fast enough to keep up with the necessary system clock frequency. It was possible, though, that on a different chip such as the Virtex series FPGAs, the on-chip RAM would have been fast enough to accomplish this.

13.2.8.2 Row/Column Order

An improvement that should be addressed was changing the order of computation from row order (left to right) to column order (up to down). The processing itself was performed in row order, which meant that results were generated in row order. While there was no ostensible problem with this in an isolated system using only Round 0, the Round 1-5 implementation for the CSPI board expected the results in column order, so this property slows the rest of the system down. The modification was relatively minor: counters in the address generator of the X1 needed to be changed; the datapath remained the same. The UCLA team investigated this issue.

13.2.8.3 Flow Control

The UCLA Round 0 system had extreme I/O requirements. The image frames were 1316 X 480 X 12bit, arriving in about 100 ms; this translated to 7.5 Mbps. Each computation FPGA generated 21-bit results (X-Y offsets) every 20 clock cycles, but only those results passing the threshold were sent. This meant that the worst case estimate for output (21 bits X 4 FPGAs/20 cycles * 60 MHz = 250Mbps) was far larger than the actual average rate. Also, by using some form of compression, this rate could certainly have been reduced.

The arrival rate of the image pixels was irregular, and the X0 was designed to capture the pixels whenever they were sent. Recall that as long as the entire image frame could be captured before the processing step was complete, there was no adverse impact on the overall cycle time. However, any interruption in processing had to be avoided because it would have lengthened the overall cycle time, which was unacceptable under the hard timing constraint. Therefore, although the flow of results could not have been halted, buffering would have been provided either on the SLAAC board FIFO, or at the CSPI board. The SLAAC-2 board had 2 FIFOs, both Xilinx XC4062 FPGAs, which together were large enough to store several thousand results, which was likely enough, based on the small number of results passing Round 0. USC/ISI worked more on this streaming data problem.

13.2.9 <u>Testing</u>

13.2.9.1 VHDL Simulations

The Round 0 system was tested using the Modelsim VHDL simulation environment 4.7f, with a full board simulation of the SLAAC-1 taking up to 10 hours for one 100ms processing cycle. The Modelsim program had the disadvantage of using no compression on the waveform files, which made it impossible to save a window of more than 20ms of waveform per simulation. Since this definitely hindered the debugging progress, it was reassuring to know that the newest versions of Modelsim finally have corrected this problem.

Another significant problem was that the automatically generated behavioral models of the Xilinx macros did not always work correctly in Modelsim. Almost all of these had to be replaced with our previously written VHDL components.

13.2.9.2 Timing Simulations

In addition to Modelsim, the Xilinx Foundation Series timing simulation tool was extensively used. This tool created back-annotated timing simulations that actually preserved the hierarchy of the original VHDL files, making the debugging process proceed much more quickly. This tool was prone to certain bugs, however, particularly those involving the behavior of bi-directional I/O pins.

13.2.9.3 Hardware Testbench

UCLA obtained a SLAAC-1 PCI board from USC/ISI to perform actual hardware tests with the Round 0 VHDL. This board was equivalent to one node of the two node SLAAC-2 board, having four reconfigurable parts: X0, X1, X2, and IF (FIFO). The X1 and X2 FPGAs on this board were Xilinx XC40150XVs, although the IR-ATR implementation used the smaller XC4085XLA. Although the SLAAC-1 PCI had no header pins for debugging, the accompanying software debugger allowed the contents of SRAM to be dumped to hard disk. Using this feature, it was possible to locate and fix several bugs. It was shown that the hardware results matched the Modelsim simulation results exactly. The fixed point implementation had a deviation from the original floating point C simulation of the Round 0 algorithm of approximately 1% ((Number of Missed Detections + Number of False Alarms) / Total Hits).

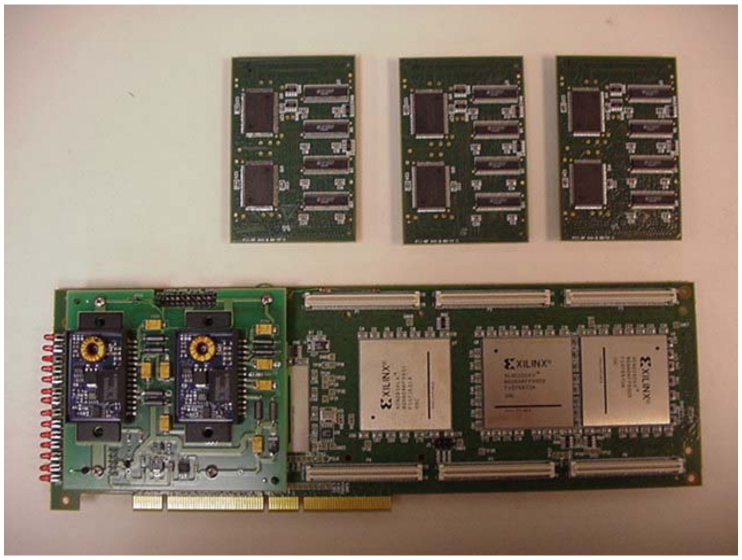


Figure 13-10 SLAAC-1 Testbench with SRAMs removed

13.2.9.4 Performance

Calculating the performance of the UCLA Round 0 FPGA design was straightforward: the number of cycles needed to complete one frame were counted and divided by the system clock frequency. The complete cycle time consisted of the time to copy the image from the X0 to the X1 and X2, and the time to process the image. Both X0 chips sent two strips in parallel, so the number of clock cycles to copy the entire image was equal to the total number of pixels of the largest strip. This value was: 1316 X 198 = 260568. Processing of the four image strips was also done in parallel (the X1a, X2a, X1b, X2b), so the number of clock cycles was equal to the total number of pixels under test within the largest strip times 20 (recall that the main datapath was divided by four pipeline stages advanced by a macroclock 20 times slower than the system clock). This number was: 1224 * 164 = 4014720. Adding these numbers together, and dividing by the system clock frequency gave the following table.

Clock Frequency*	Round 0 Cycle Time	Speed Grade **	CLB Usage (X1)
52 MHz	.082 sec	-09	1996
66 MHz	.065 sec	-07	1996

* clk frequency at maximum place & route effort
** Both parts were Xilinx XC40150XV

Table 13-1 IR/ATR Round 0 Performance

With an overall cycle time of 0.1 seconds to sustain a frame rate of 10 Hz, the UCLA Round 0 system left 0.035 seconds for the computation of Rounds 1-5. The following table shows actual computation time of software implementations for all six rounds, performed on two different platforms. This data was taken from Night Vision Labs.

Sun workstation UltraSPARC 60, network data
Total time: 13.412461 s

R0: 12.844284 s	R3: 0.074972 s
R1: 0.291631 s	R4: 0.022147 s
R2: 0.179267 s	R5: 0.000160 s

Single MVME2604, PPC 604e, network data
Total time: 19.266664 s

R0: 18.366664 s	R3: 0.183333 s
R1: 0.299999 s	R4: 0.050000 s
R2: 0.366667 s	R5: 0.000001 s

Table 13-2 IR/ATR Computation Time for Rounds 0-5

Averaging the results from both tables, Round 0 was 22 times more computationally intensive than all other rounds combined. If the hardware speedup factor UCLA achieved for Round 0 was applied to these other rounds, (speedup = 15 s / .065 s = 230 X), the total time for Rounds 1-5 would be roughly: (.29 + .36 + .18 + .05 + 0.0) / 230 = 0.0038 s. But Rounds 1-5 are left 0.035s of the 0.1 s cycle, which meant that even a speedup factor of 24 over the software implementation will be sufficient. Clearly, the full IR-ATR system could become a reality as a result of UCLA's Round 0 implementation performance gains.

13.3 Milestone Summary

13.3.1 <u>1998</u>

- For FY1998, UCLA worked on three major parts of the SLAAC project: design of the SLAAC I and II boards and their logic, the Sandia UCLA ATR and NVL IR-ATR projects, and the design and implementation of CAD tools for testing the ATR hardware.
- The bulk of the UCLA effort for this period focused on the new board being built under the Mojave program. The work on SLAAC at UCLA was still in the preliminary phases, with a ramp-up expected in the later months of CY1998.
- By mid-FY1998, the logic for the static (non-reconfiguring) FPGA was nearly finished. The static logic featured three independent sections: configuration control, results processing logic, and a processor interface. All three designs had already been separately verified, and UCLA then began simulating the entire design. The Mojave board itself was tested for electrical integrity, and UCLA successfully interfaced the board to an i960 backplane. Once the static simulations were correct, the logic was placed on the Mojave board and then tested.
- UCLA focused primarily on the board being built under the Mojave program. UCLA conducted preliminary studies for plume detection and variable bit precision for the UCLA ATR algorithm. UCLA also evaluated the NVL IR ATR problem for mapping to the Mojave board.
- Early in FY1998, UCLA studied the design methodology for the Sandia UCLA ATR system and suggested some generalizations of this methodology with respect to the algorithmic transformations. These studies were documented in two reports.
- In the first report ("Design Methodology for UCLA ATR Configurable Computing Machine", Jain and Granacki) UCLA discussed the use of an application-specific program developed by UCLA to generate HDL for the FPGA implementation of the optimized ATR algorithm.
- In the second report, ("Compiling for Adaptive Computing Systems", Granacki, Hall, Diniz, Jain) the generalizations dealt with the algorithmic transformations that were essential to optimize the implementations using configurable computing systems. UCLA concluded that one of the problems was deciding how to select the transformations for a given system. In the first quarter of FY1998, UCLA studied possible strategies to solve this problem, which were developed in conjunction with the UCLA SLAAC researchers.
- UCLA also worked on a bit-precision analysis of the NVL algorithm. Reducing the precision of the incoming image and of intermediate values was found to result in significant hardware savings and increased computational throughput. UCLA demonstrated Round 0 errors of ~5% when reducing image precision from 8-bits to 4-bits. However, the effects of introducing errors into Round 0 was unknown at that point, so it was then investigated whether or not Round 0 errors adversely affected the final results of the algorithm. The expectation was that subsequent rounds of processing would filter out false Round 0 passes. UCLA obtained C code of the entire algorithm (all rounds) from Jim Hilger and ported it to DSP Canvas. In addition, UCLA came up with

improved cost functions to analyze bit-precision tradeoffs of intermediate Round 0 values. By September 1998, the Round 0 stage was identified as the main computation bottleneck in the IR-ATR algorithm.

- An overall design concept for a 4-FPGA design was developed which was theoretically capable of meeting the timing requirements of NVL. The details of the Round 0 design concept were presented at the SLAAC meeting in Provo, UT on September 10, 1998. Work on the VHDL for the Round 0 design continued, and it was expected that the coding would be completed in FY1999, as scheduled.
- By mid-1998, the partitioning and parallelism analysis for the FPGA implementation using four FPGAs was finished. The potential to re-use a single configuration for the "hot" and "cold" computations was determined. Several blocks needed for the NVL implementation in FY99 were identified and designed, such as the carry-save adder, and register file. VHDL coding for these blocks was begun. Trade-offs between hardware implementation and system performance were studied to minimize the complexity of operations, such as division in the target FPGAs.
- Another important factor affecting the IR-ATR design process was the uncertainty of the eventual hardware platform. As a result, UCLA worked to develop a flexible design model that would enable mapping to several configurable computing boards.
- A program called WinVHDL was developed that generated VHDL for a set of configurations for the dynamic FPGA on the UCLA Mojave Board from SLD template files. Each set of template files for a target resulted in multiple FPGA configurations, such that all rotations of the SLD target were searched for. The main effort during mid-FY1998 was in developing a GUI for the compiler. The purpose of this was to capture the user input in a high-level of abstraction using terminology familiar to the ATR system designer. This input would then be translated in to parameters needed by the compiler to generate VHDL for the ATR shapesum calculation. A single user command with the GUI created the VHDL files.
- Since it was not practical to look for all rotations (templates) of one target in one FPGA configuration, some sort of partitioning was necessary in order to divide them up among the configurations. With a little study, UCLA decided that SLD templates were best, and most easily, partitioned by consecutive rotations. This became the default strategy in the compiler.
- The following three things were finished by the end of fiscal year 1998: Deliverable 1: Release FPGA application-specific CAD tool for automatic design of multi-architecture data-specific correlation hardware. Deliverable 2: Report from suitability evaluation of alternative FPGA architectures, based on the new CAD tool. Deliverable 3: New CAD tool for high performance pipelined correlation hardware.

13.3.2 <u>1999</u>

- During FY1999, work was concentrated mainly on the NVL Round 0 implementation on SLAAC boards and Los Alamos National Laboratories' (LANL) polyphase filter bank.
- By mid-FY1999, UCLA completed, and verified in simulation, the NVL IR/ATR application Round 0 algorithm targeted to the SLAAC-2 board. UCLA wrote a software

demonstration program in Visual Basic. The program automatically generated timing simulation scripts that were used by the Xilinx Foundation tools to test the placed and routed design, thereby giving truly accurate timing information as well as functional verification. UCLA worked with ISI to integrate the simulated software on the SLAAC-2 hardware. The software was extensively verified in a simulation environment at UCLA. Performance estimates based on the software timing analyzer showed a range of 80ms in the worst case, and 65ms in the best case for the full Round 0 cycle time.

- By early summer of 1999, UCLA sent the simulation software to ISI East in April for further testing. There were some issues with regard to the FIFO buffers and handshaking protocols in UCLA's code on the delivered version of the SLAAC 2 board. UCLA resolved these issues together with ISI East. This involved more extensive multiple chip simulations based on the current board design. Towards this end, UCLA ran lengthy simulations in ModelSim and modified its VHDL code based on the results to adapt its design to work properly on the ISI board.
- For the end of FY1999, the Round 0 VHDL design was tested in a full-board (multi-FPGA) Modelsim (software) testbench created by USC/ISI. The multi-chip simulation revealed bugs that had not been apparent in the single-FPGA tests performed by UCLA. In particular, there were problems with the tristating of bidirectional I/O pins, such as the SRAM data buses. In addition, code that had been written to accommodate new requirements for the method of delivering the input image data (i.e. intermittent data arrival rather than continuous stream), did not work correctly under the multi-chip simulation.
- Timing Analysis tools indicated that the total cycle time for Round 0 was approximately 78ms for the current FPGAs -09 speed grade, and 57ms if the FPGAs were upgraded to -07 speed grade. Within a month, the visible bugs were corrected and the multi-chip simulation was working correctly. The accuracy of the Round 0 VHDL simulation was observed to be 99% compared to floating point C code provided by NVL. This meant that the total number of false alarms and missed detections (all aberrations) caused by the finite precision datapath was shown to be approximately 1% of total hits for the Cota5070.arf image, strips 1 and 2. Hardware testing on the SLAAC-1 board, however, did not match the multi-chip simulation at first. Although the results for the first image frame were correct, there was more debugging work to be done as subsequent frame results were also not correct. The UCLA team attributed this to a VHDL error in the control logic and focused on fixing the problem.
- On September 20, 1999, UCLA drafted a test plan of three-month duration with the goal of debugging the multi-chip Modelsim simulation and the actual hardware implementation of Round 0 on the SLAAC-1 board. The deliverables were 6 bitstreams (3 for each SLAAC-1 on the SLAAC-2 mezzanine), accompanying VHDL, macro netlists, and documentation. The delivery date was December 20, 1999.

13.3.3 <u>2000</u>

- UCLA successfully completed its three-month test plan using the SLAAC-1 PCI board, and delivered working Round 0 bitstreams and VHDL to USC/ISI on December 20, 1999. The delivery included code and bitstreams for both nodes of the SLAAC-2 board (the SLAAC-2 board was equivalent to two SLAAC-1 nodes). USC/ISI tested the UCLA designs on the SLAAC-2 architecture. The Round 0 system using –09 speed grade XC40150XV computation FPGAs was capable of running at a clock frequency of 52 MHz, which corresponded to overall cycle time of 82ms. By upgrading to –07 speed grade FPGAs, the clock frequency was increased to 66 MHz, corresponding to an overall cycle time of 65ms.

- Early FY2000 was spent continuing to support ISI-EAST in their efforts to map the NVL Round 0 algorithm to the SLAAC-1 board. UCLA provided technical support and troubleshooting, as well as refinements of the testing procedures and documentation on the project. During the rest of this fiscal year, UCLA continued to support ISI-East in their efforts to transfer the code to the SLAAC-2 platform.

13.4 Conclusion

The UCLA implementation of the Round 0 algorithm represented a departure from the design methodology used in the creation of the Sandia UCLA ATR system. To start, the timing constraints of this system precluded the use of a run-time reconfiguration, although they may be compatible with partial reconfiguration. The system architecture for Round 0 could be described as a general purpose datapath with a template-specific address generation unit that read the image from random-access memory, whereas the Sandia UCLA ATR system architecture could be described as a template-specific datapath; a "pixel grid" which read the image as a serialized raster scan.

Such large differences in system implementation were made more interesting by the fact that both Sandia UCLA ATR and UCLA Round 0 algorithms were, at the simplest level, very similar; both algorithms used a brute force approach for correlating a template with an image. The real difference between the two lay in the nature of the templates; in the Sandia UCLA ATR case, the "sparseness" of the templates was what allowed multiple templates to be infused into a single adder tree. In the UCLA Round 0 case, the "sparseness" of the template made the building of a pixel grid/adder tree completely unfeasible.

The UCLA Round 0 design was extremely well suited to its application. The mapping of the algorithm to the SLAAC-2 board provided impressive performance: a 10 frame per second Round 0 system that operated on full 12-bit 1316X480 image frames. The implementation was extant proof that a general reconfigurable platform can meet real-world design constraints.

14 FINAL STATUS REPORT ON THE PAPMU CHALLENGE

Matthew French
University of Southern California, Information Sciences Institute

14.1 Introduction

In the final year of the SLAAC program, the Power Aware Parameter Measuring Unit (PAPMU) challenge application was added to further leverage library development done under other SLAAC challenge application, but in the context of low power. BAE Systems is conducting low-power research under the DARPA funded CSPAD and AMPS programs and has identified a suite of processing, similar to that of the SLAAC Wide-Band RF application, appropriate for ACS technology that would complete an end-to-end low power system. The goal of this challenge area is to reduce the power consumed by the parameter measuring unit ACS implementation by 10x. The focus of this research is to reduce the power utilized by intelligent FPGA algorithm mapping, device utilization, and component placement.

14.2 Technical Approach

The approach was to combine high level algorithm optimizations with libraries that optimized low-level FPGA architecture features to achieve the 10x goal in operations per watt. Improvements were to be demonstrated by first implementing the baseline PAPMU algorithms using standard FPGA and DSP implementation techniques known at the time and then comparing the power performance metrics against the implementation using the power aware algorithms and energy efficient component libraries. Due to the inefficiency in the power monitoring tools available for FPGAs at this time, power utilization was measured using Xilinx's Power Spreadsheet, Xilinx's XPower tool, and using a power monitoring testbed developed in the lab at ISI in order to gain statistical significance to the results measured.

The first step in this process was to characterize the power consumption of the various Xilinx Virtex-II micro-architecture components. Through various sample circuit designs, we found that the newer architecture features, the multiplier blocks and the block select RAM, consume two orders of magnitude more power than the traditional configurable logic block components, the shift registers, flip-flops, and look-up tables, as shown in Figure 14.1. Similarly, ISI analyzed the power performance of the routing infrastructure and found that as expected, the longer interconnects consume more power, with the long lines being about an order of magnitude worse than the rest of the interconnects, as per Figure 14.2.

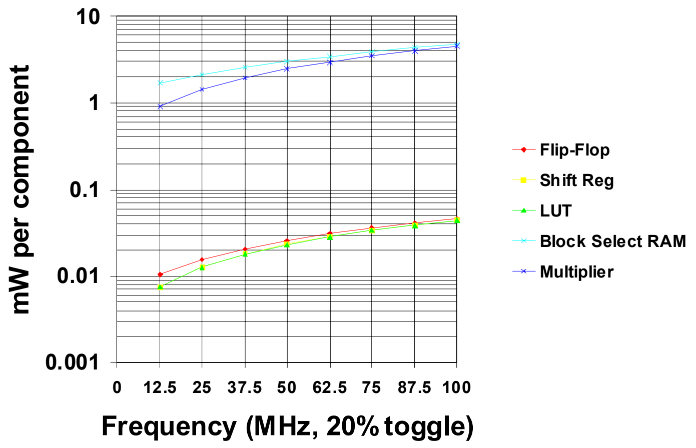


Figure 14-1 Virtex-II Component Power Utilization

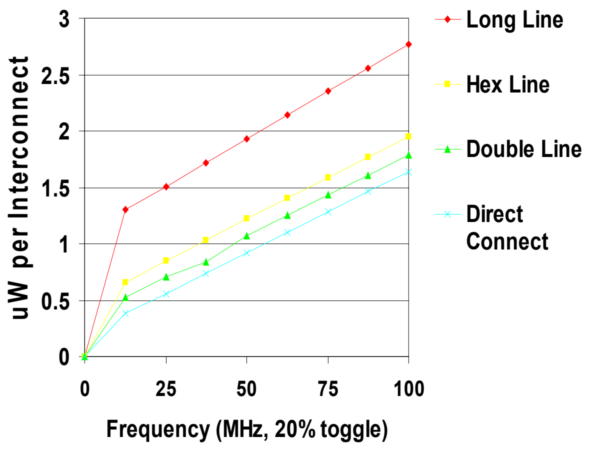


Figure 14-2 Virtex-II Interconnect Power Utilization

With this characterization information ISI profiled the PAPMU algorithm, focusing on both algorithm and architecture mapping techniques to reduce the number of block RAMs, multipliers, and long line interconnects utilized. ISI developed a library of filters for this application including, symmetric, halfband, and decimate-by-N filters that minimized the number of multipliers being used. Standard DSP techniques were used in the baseline algorithm to exploit symmetry of filter taps to reduce multipliers. These techniques were expanded on in the power aware filter library to include interleaving I and Q data processing chains, re-using adders and multipliers in filters where the data rate was a fraction of the clock rate, replacing the registers in the delay chain with more energy efficient shift registers, adding programmable filter tap coefficients to allow different types of filters to be multiplexed on the same physical resources, and using placement constraints to tightly place logic and therefore avoid costly long interconnects. Figure 14-3 shows the reduced hardware layout of a filter that exploits the known data arrival rate versus clock rate optimization. In the depicted case, data is known to arrive on every other clock sample, thereby allowing all multipliers and adders to share two coefficients. The delay chain has also been optimized from registers to more power efficient shift registers.

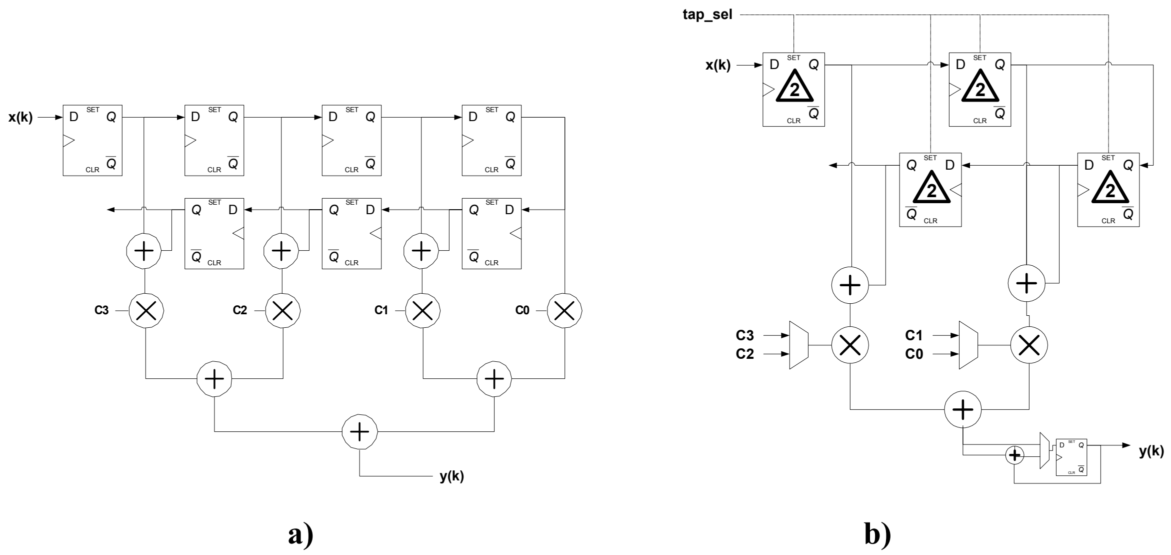


Figure 14-3 Optimized Library Filter Example
a) Traditional Halfband Filter
b) Power Efficient Halfband Filter

The filter optimizations allowed us to reduce the multipliers being used in the subband filter section of the algorithm from 108 to 28. This reduction in multipliers is enough to allow 3 full channels of the PAPMU algorithm to be implemented in the targeted Virtex-II 6000 device on the Osiris board. However, in profiling the algorithm, it was further noticed that approximately 80% of the computational time is spent in the subband tuner section and about 20% of the time is spent in the following parameter measuring section. So in order to save even more resources, an intelligent back-end parameter measuring implementation capable of multiplexing between several subband tuner inputs in a round robin fashion was created. This allowed the creation a 4-channel implementation with 4 subband tuner sections feeding a single multiplexed parameter measuring section, depicted in Figure 14-4.

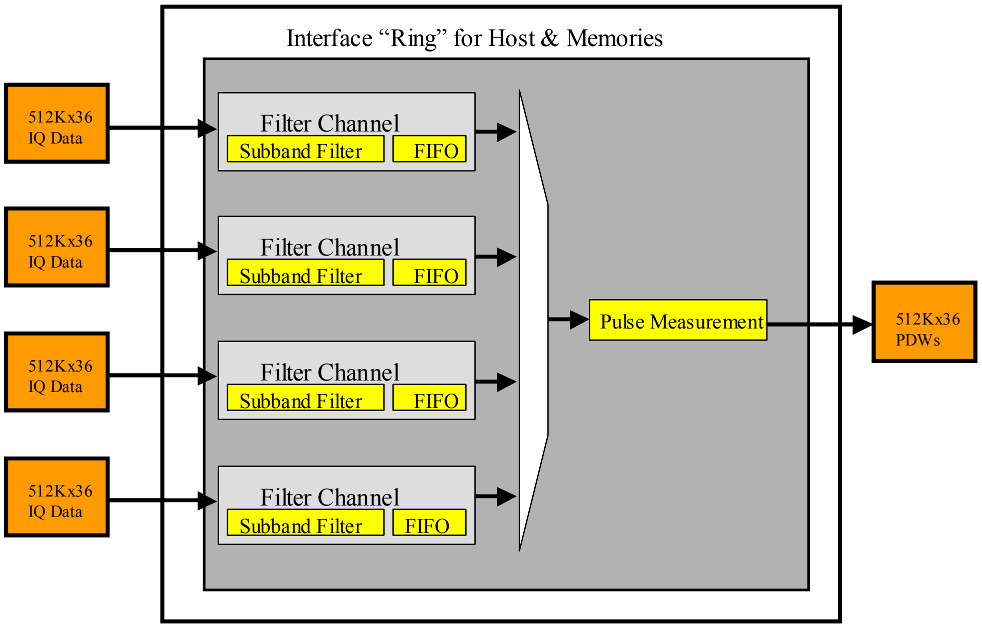


Figure 14-4 Four Channel PAPMU Implementation

Finally the last level of optimization made was in the state machine. The baseline algorithm called for a single monolithic state machine responsible for communicating with the host after each pulse was processed. In profiling the baseline implementation, it was noticed that the state machine spent 2 ms on average handshaking with the host processor after each pulse, wasting considerable energy during this time. To improve upon this a burst mode was added to the state machine, so that the host could send several pulses to the FPGA at a time and the FPGA could respond after a batch of pulses were processed. Also, rather than having a single monolithic state machine, we decoupled the state machine into several smaller state machines acting with a FIFO-like interface between different pieces of the processing. This allowed a reduction of the costly long lines being used and allowed the associated data processing logic to be placed within the FPGA much more efficiently.

14.3 Results

On June 5th, 2003 ISI gave a hardware demonstration of the baseline and optimized PAPMU application functionally working before DARPA and AFRL representatives and measured and compared the power performance. These results, along with results from power calculation spreadsheets provided by Xilinx, and estimates from the XPower tool, appear in Table 14-1. As can be seen, we easily meet our 10x goal, achieving a 54.5x improvement in operations per Watt. It is difficult to breakdown the impact of each technique, but the two biggest contributors were adding a burst mode capability to the state machine (~12-18x improvement) and reducing the multiplier usage to be able to implement 4 channels in parallel (~4x improvement).

	Baseline	Optimized	Increase
Hardware	78.8 MOps/W	4.29 GOps/W	54.5
XPower	22.3 MOps/W	1.36 GOps/W	61.0
Spreadsheets	10.5 MOps/W	651.1 MOps/W	62.0

Table 14-1 PAPMU Operations / Watt Results

This research is being utilized in ISI's new grant with NASA entitled Reconfigurable Hardware IN Orbit (RHINO), under which ISI with LANL and BYU will be developing CAD tools for low-power and space tolerant FPGA designs. At the time of this writing, no publications have been made as results have just been discovered, however we do anticipate publishing the results in one of the many FPGA related forums such as MIT LL's HPEC conference or IEEE's FCCM conference.

15 PERSONNEL

15.1 University of Southern California-Information Sciences Institute

• Robert Parker	Director, ISI-E
• John Granacki	Director, Advanced Systems Division/Researcher
• Brian Schott	Project Manager/Researcher
• Lauretta Carter	Project Manager
• Peter Bellows	Researcher
• Steve Crago	Researcher
• Joe Czarnaski	Researcher
• William Franklin	Researcher
• Matt French	Researcher
• Mary Hall	Researcher
• Dong In Kang	Researcher
• Doe-Wan Kim	Researcher
• Pankaj Topiwala	Researcher
• Terri Valenti	Researcher
• Carl Worth	Researcher
• Chen Chen	Engineer
• Jeff LaCoss	Engineer
• Scott Stansberry	Engineer
• Tam Tho	Engineer
• Li Wang	Engineer
• Ivan Hom	Student
• Jamal Faik	Visiting Research Assistant
• Venkatraman Vasudevan	Visiting Research Assistant
• Ming Zhu	Visiting Research Assistant
• Rajeev Jain	Consultant
• Elisha Lovelace	Administrator
• Susan Schneider	Administrator
• Terri Shaulis	Administrator

15.2 Brigham Young University

• Paul Graham	Research Staff
• Brad Hutchings	Faculty
• Brent Nelson	Faculty
• Doran Wilde	Faculty
• Michael Wirthlin	Faculty
• Jeremy Anderson	Student
• Mark Anderson	Student
• Dan Baker	Student
• Scott Bowden	Student
• Ben Bullough	Student
• Dan Carver	Student
• Jason Crop	Student
• Rob Franklin	Student
• Russ Fredrickson	Student
• Eric Hall	Student
• Joe Hawkins	Student
• Eric Johnson	Student
• Wes Landaker	Student
• Anshul Malvi	Student
• Aaron Martin	Student
• Jared Martin	Student
• Steve Morrison	Student
• Devin Pratt	Student
• Eric Roesler	Student
• Nathan Rollins	Student
• Michael Rytting	Student
• Matt Severson	Student
• Clark Taylor	Student
• Isaac Wagner	Student
• Xiaojun Wang	Student
• Tim Wheeler	Student
• Brian Whiting	Student
• Brett Williams	Student

15.3 University of California, Los Angeles

• John Villasenor	Faculty
• Robert Barnett	Principle/Senior Engineering Aid
• Markus Adhiwiyoyo	Senior Engineering Aid
• Jay Fahlen	Senior Engineering Aid
• Matt Fong	Senior Engineering Aid
• Aaron Schneider	Senior Engineering Aid
• Scott Siegrist	Senior Engineering Aid
• Henry Yao Chiu	Senior Engineering Aid
• Connie Yiqi Wang	Senior Engineering Aid
• Vipin Aggarwal	Staff Research Associate
• Mike Severa	Staff Research Associate
• Christopher Jones	Development Engineer
• Jeffrey Tseng	Assistant Development Engineer
• Adam Li	Postdoctoral Researcher/ Staff Research Engineer
• Wesley Negus	Research Student/Graduate Student Researcher
• Dan Benyamin	Graduate Student Researcher
• Samson Ho	Graduate Student Researcher
• Ksenija Lakovic	Graduate Student Researcher
• Dannie Lau	Graduate Student Researcher
• Raghu Rao	Graduate Student Researcher
• Matthieu Tisserand	Graduate Student Researcher
• Roland Osborne	Student
• Arthur Chang	Lab Assistant

15.4 Sandia National Laboratories

• Larry Hostetler	Director/ Senior Manager
• Drayton Boozer	Manager
• Wallace Bow	Researcher
• Brian Bray	Researcher
• Denise Carlson	Researcher
• Doug Doerfler	researcher
• Joe Fogler	Researcher
• Richard Meyer	Researcher
• Katherine Simonson	Researcher

15.5 Los Alamos National Laboratory

• Maya Gokhale	PI/ Researcher
• Kevin McCabe	Co-PI/Researcher
• Joe Arrowood	Researcher
• Jeff Bloch	Researcher
• Bryan Burke	Researcher
• Mark Dunham	Researcher
• Herb Fry	Researcher
• Jan Frigo	Researcher
• Dominique Lavenier	Researcher
• Tony Nelson	Researcher
• Doug Patrick	Researcher
• Reid Porter	Researcher
• Scott Robinson	Researcher
• Anthony Salazar	Researcher
• John Szymanski	Researcher
• James Theiler	Researcher

15.6 Virginia Tech

• Peter Athanas	Faculty
• Mark Jones	Faculty
• Lijia Chen	Graduate Research Assistant
• Dennis Collins	Graduate Research Assistant
• James Hendry	Graduate Research Assistant
• Heather Hill	Graduate Research Assistant
• David Lehn	Graduate Research Assistant
• Zahi Nakad	Graduate Research Assistant
• Will Worek	Graduate Research Assistant
• Matthew Yaconis	Graduate Research Assistant
• Kuan Yao	Graduate Research Assistant
• Zhenping Liu	Graduate Research Assistant
• Emad Ibrahim	Undergraduate Research Assistant
• Christian Laughlin	Undergraduate Research Assistant
• George Morgan	Undergraduate Research Assistant
• Lucas Scharf	Undergraduate Research Assistant
• Jonathan Scott	Undergraduate Research Assistant
• Lucas Scharf	Undergraduate Research Assistant
• John Shiflett	Undergraduate Research Assistant

15.7 Lockheed Martin Government Electronic Systems

• Robert Graybill	Director
• Jay Hansen	Director (Acting)
• Rick Pancoast	Advanced Digital Processing Center Staff
• Ray DiFelice	Advanced Processor Business Development
• Mike McCloskey	Manager, Radar Equipment Development
• Paul Ramondetta	Principal Member Eng. Staff, Radar Equip. Development
• Paul Delrocini	Lead Member Engineering Staff
• Gregory Coxson	Senior Member Engineering Staff
• Walter Mazur	Senior Member Engineering Staff
• Kenneth Walling	Principal/Senior Member Engineering Staff
• William Strack	Senior Member Engineering Staff
• Jeanine Matthews	Program Management
• Thomas Smith	Program ManagementAppendix

15.8 BAE Systems

• Dale Rickard	Technical Director
• Robert Basset	Lead Member Engineering Staff
• Bill Grizzle	Senior Member Engineering Staff
• Jeff Robertson	Senior Member Engineering Staff

16 APPENDIX

16.1 List of Acronyms

ACS- Adaptive Computing Systems
ACS API – Adaptive Computing Systems Application Programming Interface
AEGIS AN/SPY-1 – AEGIS class cruiser radar component
API – Application Programming Interface
ARTM - ATR Relational Template Matching program
ASAPP- Accelerating Segmentation and Pixel Purity Index
ASIC – Application Specific Integrated Circuit
ATR- Automatic Target Recognition
BGA – Ball Grid Array
C4PL -CYTO Portable Parallel Picture Processing Language
CAD – Computer Aided Design
CC - Configuration Controller
CDI - Contamination Distribution Indexer
CIC - Cascaded Integrator-Comb
CLBs – Control Logic Blocks
CORDIC - Coordinate Rotation Digital Computer
COTS - Commercial off-the-shelf
CSPI - CSPI, Inc.
CYTO – image processing computer
DAPS - Deployable Adaptive Processing Systems
DFT- Discrete Fourier Transform
DMA - Direct Memory Access
DRP - Deployable Reference Platform
DSP – Digital Signal Processor
ECMA - Electronic Countermeasures Assessment
EEPROM – Electrically Erasable Programmable Read Only Memory
ELINT – Electronic Intelligence
ERIM- Environmental Research Institute of Michigan
FFT- Fast Fourier Transform
FIFO – First In First Out
FIR - Finite Impulse Response
FOA- Focus Of Attention
FPGA - Field Programmable Gate Array
GTS - Global Tri State
HDL – Hardware Description Language
HIP- Hyperspectral Image Processing
HIRIS – Hyperspectral Infrared Imaging System
HPC – High Performance Computing
IR-ATR -Infrared Automatic Target Recognition
JHDL – registered trademark, Brigham Young University
LED – Light Emitting Diode
LMGES-Lockheed Martin Government Electronic Systems

MAC - Multiply Accumulate
MDIP – Multi-Dimensional Image Processing
MPI - Message Passing Interface
NIMA - National Imagery and Mapping government agency
NIS - Nonproliferation and International Security
NSA – National Security Agency
NUWC - Naval Undersea Warfare Center
NVL - Night Vision Laboratories
PAPMU – Power Aware Parameter Measuring Unit
PCI – Peripheral Component Interconnect
PCI DMA - Peripheral Component Interconnect Direct Memory Access
PCLK- Processor Clock
PEO- Program Executive Office
PMC - PCI Mezzanine Card
POOKA- Accelerated Image Processor using Machine Learning
POSIX - Portable Operating System Interface Extensions
PPI - Pixel Purity Index
RFIP- Rapid Feature Identification Project
RPM - Relationally Placed Macros
RRP - Research Reference Platform
RTL Register Transfer Language VHSIC
RTR- run-time reconfiguration
SAN - System Area Network
SAR - Synthetic Aperture Radar
SCSI – Small Computer System Interface
SIMD – Single Instruction Multiple Data
SLAAC - System Level Applications of Adaptive Computing
SLD- Second Level Detection
SMP – Symmetric Multiprocessors
SODIMM - Small Outline Dual In-line Memory Module(s)
SONAR – Sound Navigation and Ranging
SRAM – Synchronous Random Access Memory
SRT Sweeny-Robertson-Tocher
STARS – Surveillance and Target Attack Radar System
SUIF - Stanford University Intermediate Format
ToP - Tower of Power
TSC- Theater Surface Combatants
UAV – Unmanned Airborne Vehicle
USAF – US Air Force
VHDL - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
VME/VXI - Versa Module Eurocard / VME eXtensions for Instrumentation
XML - eXtensible Markup Language
ZBT SRAMs - Zero-Bus Turnaround Synchronous Random Access Memory

16.2 List of Related Publications

Pg. 113, Miriam Leeser, Pavel Belanovic, Michael Estlick, Maya Gokhale, John Syzmanski, James Theiler, "Applying Reconfigurable Hardware to the Analysis of Multispectral and Hyperspectral Imagery," SPIE 2001.

Pg. 121, R. Porter, K. McCabe, N. Bergman, “An Applications Approach to Evolvable Hardware,” submitted to the NASA/DoD Workshop on Evolvable Hardware, July 1999.

Pg. 126, Scott Hemmert and Brad Hutchings, “"An Application-Specific Compiler for High-Speed Binary Image Morphology" Symposium on Field-Programmable Custom Computing Machines, Napa, CA, April, 2001.

Pg. 137, Brian Schott and Chen Chen and Steve Crago and Joe Czarnaski and Matt French and Ivan Hom and Tam Tho and Terri Valenti, Architectures for System-Level Applications of Adaptive Computing, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 270-271, IEEE Computer Society Press, April 1999.

Pg. 139, J. Theiler, J. Frigo, M. Gokhale, and J. J. Szymanski. “Co-design of Software and Hardware to Implement Remote Sensing Algorithms.” Proc. SPIE 4480 (2001) 86--99. {Note}: Invited Paper

Pg. 153, Tim Grembowski and Roar Lien and Kris Gaj (HP) and Nghi Nguyen and Peter Bellows and Jaroslav Flidr and Tom Lehman and Brian Schott (HP), Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512, Lecture Notes in Computer Science, Vol. 2433, p. 75-??, 2002.

Pg. 168, Brent Nelson "Configurable Computing and SONAR Processing - Architectures and Implementations.” November 2001.

Pg. 173, K.-N. Chia, H. J. Kim, S. Lansing, W. H. Mangione-Smith, and J. Villasenor, “Configurable Computing Solutions for Automatic Target Recognition,” IEEE Transactions on VLSI, vol. 6, pp.364-371, 1988.

Pg. 183, J. Theiler, M. Leeser, M. Estlick, and J. J. Szymanski. “Design Issues for Hardware Implementation of an Algorithm for Segmenting Hyperspectral Imagery." Proc. SPIE 4132 (2000) 99--106.

Pg. 191, Maya Gokhale, Jan Frigo, Kevin McCabe, James Theiler, Dominique Lavenier, "Early Experience with a Hybrid Processor: K-Means Clustering," ERSA 2001.

Pg. 198, Jan Frigo, Maya Gokhale, Dominique Lavenier, “Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective," FPGA 2001.

Pg. 205, Reid Porter, Maya Gokhale, Neal Harvey, Simon Perkins, Cody Young, "Evolving Network Architectures with Custom Computers for Multi-Spectral Feature Identification," Third NASA/DoD Workshop on Evolvable Hardware, 2001

Pg. 215, Pawel Chodowiec, Kris Gaj, Peter Bellows, Brian Schott: Experimental Testing of the Gigabit IPSec-Compliant Implementations of Rijndael and Triple DES Using SLAAC-1V FPGA Accelerator Board. ISC 2001: 220-234

Pg. 230,Graham, Paul, and Brent Nelson. FPGA-Based Sonar Processing. Brigham Young University, 1997.

Pg. 242, D. Lavenier, J. Theiler, M. Gokhale, J. Frigo, and J.J. Szymanski. “FPGA Implementation of the Pixel Purity Algorithm for Hyper-Spectral Images." Proc SPIE 4212 (2000) 30--41.

Pg. 257, Graham, Paul, and Brent Nelson. “FPGA’s and DSP’s for Sonar Processing-Inner Loop Computations”. Diss. Brigham Young U, 1998.

Pg. 267, "GIGA OP DSP on FPGA" by Brad Hutchings and Brent Nelson. This paper was presented at ICASSP 2001 held in Salt Lake City in May 2001. Part of it describes results obtained in the ATR part of the SLAAC project and part of it focuses on the SONAR work.

Pg. 271, Zahi Nakad, “High Performance Applications on Reconfigurable Clusters,” M.S. thesis, Virginia Tech, 2000.

Pg. 341, Mark Jones, Lucas Scharf, Jon Scott, Christian Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas, and Brian Schott, “Implementing an API for Distributed Adaptive Computing Systems,” Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, April, 1999.

Pg. 354, Kuan Yao, “Implementing an Application Programming Interface for Distributed Adaptive Computing Systems,” M.S. thesis, Virginia Tech, 2000.

Pg. 464, Michael J. Wirthlin, Steve Morrison, Paul Graham and Brian Bray “Improving the Performance and Efficiency of an Adaptive Amplification Operation Using Configurable Hardware,” IEEE Symposium on Field Programmable Custom Computing Machines (FCCM), 2000.

Pg. 473, William Worek, “Matching Genetic Sequences in Distributed Adaptive Computing Systems,” M.S. thesis, Virginia Tech, 2002.

Pg. 576, B. Schott, S. Crago, C. Chen, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, “Reconfigurable Architectures for Systems-Level Applications of Adaptive Computing,” submitted to VLSI Design special issue on reconfigurable computing.

Pg. 592, P. Graham and B. Nelson, “Reconfigurable Processors for High-Performance, Embedded Digital Signal Processing, in Field-Programmable Logic and Applications. Proceedings of the 9th International Workshop, FPL '99, pages 1-10.

Pg. 602, Stephen P. Crago and Brian Schott and Robert Parker (HP), SLAAC: A Distributed Architecture for Adaptive Computing, IEEE Symposium on FPGAs for Custom Computing Machines, pp. 286-287, IEEE Computer Society Press, April 1998.

Pg. 604, J. Theiler, D. Lavenier, N.R. Harvey, S.J. Perkins, and J.J. Szymanski. “Using blocks of skewers for faster computation of Pixel Purity Index." Proc. SPIE 4132 (2000) 61--71.

Applying Reconfigurable Hardware to the Analysis of Multispectral and Hyperspectral Imagery

Miriam Leeser[1], Pavle Belanovic[1], Michael Estlick[1],
Maya Gokhale[2], John J. Szymanski[2], and James Theiler[2]

[1]Department of Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts 02115

[2]Space and Remote Sensing Sciences Group
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

ABSTRACT

Unsupervised clustering is a powerful technique for processing multispectral and hyperspectral images. Last year, we reported on an implementation of k-means clustering for multispectral images. Our implementation in reconfigurable hardware processed 10 channel multispectral images two orders of magnitude faster than a software implementation of the same algorithm. The advantage of using reconfigurable hardware to accelerate k-means clustering is clear; the disadvantage is the hardware implementation worked for one specific dataset. It is a non-trivial task to change this implementation to handle a dataset with different number of spectral channels, bits per spectral channel, or number of pixels; or to change the number of clusters. These changes required knowledge of the hardware design process and could take several days of a designer's time.

Since multispectral data sets come in many shapes and sizes, being able to easily change the k-means implementation for these different data sets is important. For this reason, we have developed a parameterized implementation of the k-means algorithm. Our design is parameterized by the number of pixels in an image, the number of channels per pixel, and the number of bits per channel as well as the number of clusters. These parameters can easily be changed in a few minutes by someone not familiar with the design process. The resulting implementation is very close in performance to the original hardware implementation. It has the added advantage that the parameterized design compiles approximately three times faster than the original.

Keywords: hyperspectral, k-means, image processing, algorithm, field-programmable gate array (FPGA)

1. INTRODUCTION

Modern hyperspectral imagers can produce data cubes with hundreds of spectral channels and millions of pixels. One way to cope with this massive quantity is to organize the data so that pixels with similar spectral content are clustered together in the same category. This provides both a compression of the data and a segmentation of the image that can be useful for other image processing tasks downstream.

The classic approach for segmentation of multidimensional data is the k-means algorithm; this is an iterative method that produces successively better segmentations.[1,2] It is a simple algorithm but the computational expense can be considerable, particularly for clustering large hyperspectral images into many categories. Last year we reported on results of the ASAPP (Accelerating Segmentation and Pixel Purity) project in implementing the k-means algorithm in field-programmable gate array (FPGA) hardware.[3]

The standard software implementation of k-means uses floating-point arithmetic and Euclidean distances. By fixing the precision of the computation and by employing the Manhattan distance metric, we were able to fit more distance-computation calculations on the chip, obtain a higher degree of fine-grain parallelism, and therefore faster performance. The price of these optimizations was slightly less optimal clusters. We reported on the use of different

Send correspondence to Miriam Leeser: E-mail: mel@ece.neu.edu distance metrics and of different levels of precision truncation, and their effect on total within-class variance of a cluster.[3] The hardware implementation presented was designed for 10 channel, 12 bits per channel multispectral images based on MTI datasets. Our results showed that the hardware implementation of k-means ran two orders of magnitude faster than the software version, with an average increase in within-class variance of about 3 percent.

The major drawback of the hardware implementation is that it required in-depth knowledge of hardware design to accomplish. In addition, it works only for a dataset with 10 channels per pixel, 12 bits per channel, and 8 clusters. These parameters were hardwired into the design in many different locations. In addition, a change in one parameter has other effects such as requiring a wider accumulator in another part of the design. Changing these values requires knowledge of hardware design, and may take several days to accomplish.

Multispectral data sets come in many shapes and sizes.[4,5] Our goal is to make the k-means hardware implementation more accessible to an image analyst by making it easier for the analyst to adapt the implementation to different data sets. This translates to the requirement of making the parameters easier to change. An analyst may wish to change the number of clusters, or apply k-means to a different dataset. In this paper, we present a parameterized k-means implementation that allows the user to easily change the parameters without requiring knowledge of the implementation.

The motivation behind parameterizing the k-means algorithm is the vastly different datasets available. For example, MTI, IKONOS and Landsat images have different numbers of spectral bands, pixels, etc. The 10 channel images we use are derived from the AVIRIS data to mock up MTI data. Real MTI data[6] has 4 visible and 12 IR bands, so sometimes an analyst wished to process 4 channels of data, sometimes 12 and sometimes 16. A typical MTI image has roughly 1200×3200 pixels for visible imagery (5m ground resolution), and 300×800 for IR imagery (20m resolution). A number of other sensors are summarized by NASA.[7] IKONOS imagery is 4 spectral bands (R,G,B,NIR) + 1 panchromatic band; it has 1m resolution on the ground. Images are 1000×1000 pixels. Landsat 7 has eight bands: 6 spectral, 1 thermal, and 1 panchromatic. Ground resolution varies from 30m for spectral, 60m for thermal, and 15m for panchromatic. A typical image size is 183×170 km, or over $5K \times 5K$ pixels.[8] Many analysts use subimages when processing this data. Multspectral images come from other sources as well. For example, we have some medical data of cancerous tissue that is 705×601 pixels and 24 channels. There are reasons for clustering each of these data sets. This requires varying the parameters specified.

2. K-MEANS CLUSTERING

Given a set of N pixels, each composed of C spectral channels, and represented as a point in C-dimensional Euclidean space (that is, $\mathbf{x}_n \in \mathcal{R}^C$, with $n = 1, \ldots, N$); we partition the pixels into K clusters with the property that pixels in the same cluster are spectrally similar. Each cluster is associated with a "prototype" or "center" value which is representative of (and close to) the pixels in that class.

One measure of the quality of a partition is the within-class variance; this is the sum of squared (Euclidean) distances from each pixel to that pixel's cluster center.

For a fixed partition, the optimal (in the sense of minimum within-class variance) location for each center is the mean of all pixels in each class. And for a fixed choice of centers, the optimal partition assigns points to the cluster whose centers are closest. The k-means clustering algorithm provides an iterative scheme that operates over a fixed number (K) of clusters, while attempting to simultaneously optimize center locations and pixels assignments.

From an initial sampling, the algorithm loops over all the data points, and reassigns each to the cluster whose center it is closest to. After the pass through the data, the cluster centers are recomputed. Each iteration reduces the total within-class variance for the clustering, so it is guaranteed that after enough iterations, the algorithm will converge, and further passes will not reassign points. It bears remarking that this is a local minimum.

There are many variants of the basic k-means clustering algorithm. In some variants, the number of clusters (K) is altered as part of the algorithm.[9] Other variants of k-means update the cluster centers each time a point is reassigned to a new cluster. Others vary in the way initialization is performed and termination is determined.

We have chosen a k-means algorithm that is particularly well-suited to hardware implementation. K is chosen by the image analyst before the implementation is generated. The algorithm runs by first assigning every point to a new cluster (a complete iteration through all the pixels in the image) and second computing the new cluster centers. Initialization and termination are determined in software, allowing for many different options. For example, the implementation may terminate after no points are reassigned or after a fixed number of iterations. These are standard variants on the basic k-means algorithm.

3. EFFICIENT K-MEANS ALGORITHM FOR HARDWARE IMPLEMENTATION

Our implementation includes other variations that are seldom found in software implementations, but result in more efficient hardware implementation. These include using a different distance metric, and minimizing bitwidths of the input data.

Points are assigned to the cluster centers to which they are closest; for the minimum within-class variance criterion, "closest" is defined in terms of the Euclidean distance. Consider a point $\mathbf{x}$ and cluster center $\mathbf{c}$ where i indexes the spectral components of each. The Euclidean distance is defined as:

$$\|\mathbf{x}-\mathbf{c}\|^2 = \sum_i |x_i - c_i|^2. \tag{1}$$

But other distance measures can also be used; for instance, the general family of p-metrics (for which the Euclidean distance is the special case $p = 2$) is given by:

$$\|\mathbf{x}-\mathbf{c}\|^p = \sum_i |x_i - c_i|^p. \tag{2}$$

To perform a k-means iteration, one must compute the distance from every point to every center. If there are N points, K centers, and C spectral channels, then there will be $O(NKC)$ operations. For the Euclidean distance, each operation requires computing the square of a number.

The Euclidean distance has several advantages. For one, the distance is rotationally invariant. Furthermore, minimizing the Euclidean distance minimizes the within-class variance. On the other hand, the Euclidean distance is much more expensive to implement in hardware. The Manhattan distance, corresponding to $p = 1$, is the sum of absolute values of the coordinate differences; the Max distance, corresponding to $p = \infty$ is the maximum of the absolute values of the coordinate differences. Both these alternative distance metrics were investigated because neither requires any multiplication. In addition, a linear combination of the two was considered.

Data independent and data dependent experiments showed that the Manhattan distance could be used with a small loss in quality of the resulting clusters. The resulting hardware is much more compact than that using the Euclidean distance. This translates to significant speedup in run-times since more parallelism can be implemented.

Another area we investigated was minimizing bitwidths to keep the hardware implementation compact and to increase parallelism. In hardware implementations, the number of bits can be chosen to optimize the size of the hardware design. An advantage of using the Manhattan distance is that the growth in bitwidth in the datapath is smaller than that generated by the Euclidean distance, since the squared values in the Euclidean distance produce twice the bitwidth.

In addition, we investigated the number of bits per channel that influenced the outcome of clustering. Since a large amount of data is being compressed to a very small number of clusters, we hypothesized that much of the input data was not contributing to the result. This hypothesis was borne out in both data independent and data dependent experiments. For AVIRIS images, half the input bits per channel could be ignored with a very small loss in quality, provided a few extra bits were used to represent the cluster means.[10] The result of doing this is the generation of more efficient hardware. The analyst must be able to easily change the number of bits per channel in the implementation to take advantage of this optimization.

4. K-MEANS HARDWARE IMPLEMENTATION

The k-means clustering algorithm consists of two steps: (1)assigning each pixel in the image to one of the K clusters, and (2) recomputing the cluster centers. In order to recompute the cluster centers, not only must the number of pixels assigned to a cluster be computed, but the values of all pixels assigned to each cluster must be accumulated on a channel by channel basis. Our hardware implementation includes step 1 of the algorithm as well as all the accumulation. Cluster centers are computed on the host, but the image data is accessed only by the hardware. Hence, inputs to the hardware implementation are all pixels in the image and the values of all K clusters; outputs are cluster assignments for each pixel and accumulated values for each cluster.

Our design, shown in Figure 1, is fully pipelined. Each clock cycle, a new input pixel is read from memory. The hardware consists of a datapath for computing the nearest cluster center, a pixel shift register to keep the pixel values

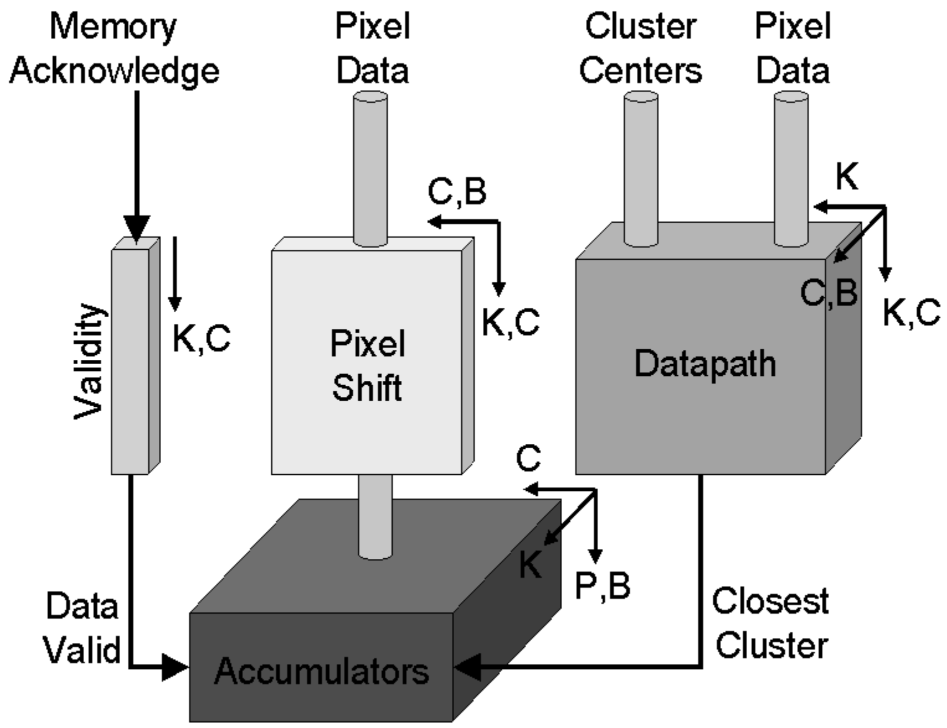


Figure 1. K-means Hardware Design

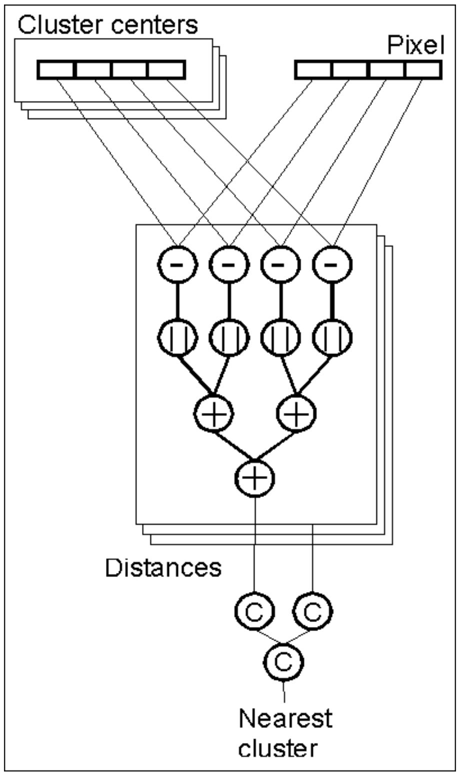


Figure 2. Datapath

with their corresponding cluster center, hardware for keeping track of the validity of the output, and an accumulation block for accumulating the pixel values for pixels assigned to each cluster.

The datapath section of the design assigns each pixel to the closest cluster. One pixel is read from the memory at each clock cycle and placed at the input to the datapath. All spectral channels of the pixel appear at the input to the datapath simultaneously. The other input to the datapath circuit is the center of each of the K clusters. The output of the datapath circuit is a cluster number, indicating which of the K clusters the pixel was assigned to. The Manhattan distance is used.

Figure 2 shows the datapath circuit in more detail. First the distance is calculated in parallel between the pixel and each of the K clusters. Calculating the Manhattan distance involves three operations performed serially: subtraction, absolute value and addition. Then, the results of these K distance calculations are compared to find the minimum. Both the addition of channel distance calculations and the comparison of the total distance calculation require tree structures to compute. The result is the number of the cluster center closest to the pixel.

The accumulators add the pixel value, on a channel by channel basis, to the cluster to which the pixel is being assigned. This requires the output of the datapath circuit as well as the pixel value, which was retrieved from memory for the datapath calculation. The pixel value must appear at the input to the accumulator at the same time as the cluster number to which the pixel is assigned. This is accomplished by the pixel shift circuit, which delays the pixel data the number of pipeline stages of the datapath. The number of pipeline stages depends on both the number of clusters and the number of channels due to the tree structure of the comparison and distance calculation in the datapath.

The validity pipeline is a similar structure to the pixel shift circuit. At the beginning of processing, pixel data is requested from memory. Once data is available, the memory indicates that the data is valid, which triggers the datapath and pixel shift circuits to begin operation. The accumulator circuit must be triggered to begin operation at the end of the datapath operation. This is accomplished by delaying the memory data validity signal the appropriate number of clock cycles.

Pixel values are added to the assigned cluster's accumulated values by the accumulator circuit. The three inputs to the accumulator circuit are the enable signal from the validity pipeline, the pixel value from the pixel shift circuit, and the cluster number from the datapath circuit. Each of the K accumulators has associated with it a counter, keeping track of how many pixels have been added. At the end of a complete iteration of the image, the values of all the accumulators and counters contain the information required to compute the new cluster centers. The cluster numbers for all the pixels processed are stored in memory.

5. PARAMETERIZATION

The size of the design depends on the amount and organization of the data that needs to be processed as well as on the number of clusters. The basic parameters that define the design are K, C, B and P. K is the number of clusters, C is the number of channels per pixel, and B is the number of bits per channel. N is the total number of pixels in the image, and P is a summary of the number of pixels used to parameterize the hardware design. P is defined as: $\lceil \log_2(N) \rceil$. For example, an image with 1024×1024 pixels has $N = 1048576$ and $P = 20$. For an MTI IR band with 300×800 pixels, $N = 240000$ and $P = 18$. The parameters to our implementation are summarized in Table 1.

Parameter	Definition	Source
K	number of clusters	Image Analyst
C	number of channels per pixel	Image
B	number of bits per channel	Image
N	total number of pixels	Image
P	$\lceil \log_2(N) \rceil$	Image

Table 1. Parameters for K-means

To parameterize the design, we mapped its growth in terms of these basic parameters. This applies to the four sub-circuits: datapath, pixel shift, validity pipeline, and accumulator, as well as the signals connecting them.

The datapath circuit is pipelined through D stages. D is defined as $2 + \lceil \log_2(C) \rceil + \lceil \log_2(K) \rceil$. The dependence on the number of channels is due to the adder tree that accumulates the distances over channels; the dependence on K is due to the comparator tree to choose the closest cluster. The complexity of the datapath circuit is parameterized by K as well. There are K distances being calculated in parallel and thus K individual pipelines need to be implemented. The bitwidth of the datapath circuit scales with parameters C and B because each of the pipelines for calculating distance depends on the bitwidth B for the size of subtractors, complementers (for absolute value) and adders, as well as C for the size of the addition tree used to sum all the channels.

The depth of the pixel shift circuit is D, the same as that of the datapath. The width of the pixel shift circuit is naturally characterized by the size of the pixel data that ripples through it. Hence, the width scales according to the number of channels C and the the number of bits in each channel B. The validity pipeline is a cascade of 1-bit registers and thus uses only the depth dimension D.

The accumulator circuit must take into consideration the total number of pixels being processed. The size of the circuit is characterized by the number of channels C because each channel of the pixel is accumulated separately and all channels are processed in parallel. The width of the accumulator circuit scales by K because there is one accumulator in the circuit for each cluster. The depth of the circuit, depends on parameters B and P because each register that accumulates one channel of the pixel varies in size with the number of bits in the channel B, and the total number of bits that can potentially be accumulated into the register. The registers of the accumulators are sized for the worst-case scenario in which all the pixels are assigned to one cluster and every channel of every pixel contains the maximum possible value.

6. RESULTS

Our designs are implemented on a Wildstar PCI board from Annapolis Microsystems.[11] The board has three Xilinx Virtex 1000 FPGAs[12] and 40MB of SRAM. The Wildstar is in a 500MHz Pentium III workstation. The k-means implementations we present use one Virtex 1000. There are two implementations. The first, called the RTL design, is the design presented last year.[3] The second is the parameterized version of the design we have presented in this paper. Both implementations were written in VHDL and synthesized with Synplicity's Synplify and Xilinx Alliance tools. The hardware and software used is commercially available.

The RTL design was implemented in VHDL as a set of processes and uses Synplicity design tools[13] plus Xilinx place-and-route tools[14] to generate a bitstream. The parameterized design also makes use of Synplicity design tools, but creates the design out of components built using the Xilinx CoreGen libraries.[15] Xilinx place-and-route tools are once again used to generate the bitstream. Once the bitstream has been created, it is downloaded to the board, resulting in operational hardware.

Both designs are fully pipelined; one pixel is processed every clock cycle. These implementations assume that a single pixel ($C \times B$ bits) can be read in one clock cycle. For the Annapolis Wildstar board, this translates into a constraint of $C \times B \leq 128$ because 128 bits can be accessed from memory in one clock cycle. To simplify the design, we have also constrained the number of classes to being a power of two.

The design presented last year classifies a 1024×1024 pixel image with 10 channels of 12 bit data per channel into 8 clusters. We compare the performance of the RTL design to the performance of the parameterized design in Table 2. We are interested in area and runtime of the two designs, as well as the place-and-route time, which is the time to create the hardware design.

Design	RTL	Parameterized
Area in slices	9420	8884
Percent area used	76 %	72 %
Total gate equivalents	406,536	433,474
Max Frequency	63.78Mhz	63.07Mhz
Place-and-route time	3:32:56	1:04:49

Table 2. Comparison of RTL and Parameterized Designs

In our results, area is measured by the number of logic slices used on the FPGA. The Virtex 1000 chips have a total of 12288 slices. Typically, some slices are used to route signals instead of as logic. Another measure of area is the number of "gate equivalents" used to implement the design. A gate equivalent is a two-input logic gate; this is a standard measure of area used in digital hardware design. The measure of speed used is the maximum clock frequency. These numbers are generated by the Xilinx place-and-route tools.

The two designs generated with these competing design styles have comparable area and speed. The parameterized design has more gate equivalents packed into fewer Virtex slices; we believe this is due to the fact that the components used in the parameterized design make more efficient use of hardware resources. The parameterized design uses designed and optimized modules provided by the Xilinx CoreGen tools; the RTL design requires the synthesis tool to generate these modules.

The largest difference between the two designs is the ease of generating new implementations using the parameterized methodology, and the fact that the parameterized design has a place-and-route time more than three times faster than the RTL design without a loss of quality in the results. To illustrate the ease of changing parameters, we have implemented k-means clustering on the Wildstar board for the combinations of parameters shown in Table 3.

K	C	B	P
16	5	10	20
8	14	8	20
8	20	6	20
8	10	12	20

Table 3. Combinations of Parameters for which K-means Designs have been Generated

Each new design took a few minutes of setting the parameters plus approximately an hour to place-and-route. Changing the RTL design to generate a new design takes several days of changes as well as three and a half hours of place-and-route time. In addition, it is much easier to introduce errors when changing the RTL code versus the parameterized code.

While these results reflect the use of one Virtex 1000 chip on a Wildstar board, in reality we parallelize the implementation by using two Virtex chips with each processing half the image. This change affects the host code only, and the speedup is the same for both the RTL code and the parameterized code.

7. CONCLUSIONS

In order to accommodate real data sets, we have implemented parameterized k-means in reconfigurable hardware. Our implementation is parameterized by the number of clusters K, the the number of channels, the number of bits per channel and the number of pixels in the image. The performance of the parameterized k-means implementation is very close in area and speed to the non-parameterized version. The main advantage, in addition to the parameterization, is that it generates a hardware solution three times faster. The reason for this speedup is the use of predesigned macros for components such as adders and comparators.

Our current approach is constrained by the memory bandwidth to the Virtex chip on the Wildstar board. We process one pixel each clock cycle, and fully pipeline the processing. In the future, we plan to investigate adding pipelining to our parameterized implementation. In other words, the structure of the pipeline would change depending on the number of clock cycles required to fetch the input data. In addition, we are investigating applying this parameterized approach in other domains, including parameterized floating point modules for reconfigurable hardware.

ACKNOWLEDGMENTS

We would like to thank Annapolis Microsystems for their support, and Synplicity and Xilinx Corporations for their donations to the Rapid Prototyping Laboratory at Northeastern University. We would like to thank researchers at Los Alamos and at Northeastern University who contributed to this project. This research was funded by the US Department of Defense and Department of Energy.

REFERENCES

1. G. B. Coleman and H. C. Andrews, "Image segmentation by clustering," *Proceedings of the IEEE* **67**(5), pp. 773–785, 1979.
2. A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computing Surveys* **31**, pp. 264–323, 1999.
3. J. Theiler, M. Leeser, M. Estlick, and J. J. Szymanski, "Design issues for hardware implementation of an algorithm for segmenting hyperspectral imagery," in *Image Spectrometry VI*, *Proc. SPIE* **4132**, pp. 99–106, 2000.
4. J. R. Jensen, *Introductory Digital Image Processing: A Remote Sensing Perspective*, Prentice Hall, 1996. Second Edition.
5. R. A. Schowengerdt, *Remote Sensing: Models and Methods for Image Processing*, Academic Press, 1997. Second Edition.
6. Los Alamos National Laboratory, 2000. `http://nis-www.lanl.gov/nis-projects/mti/`.
7. NASA, 2000. `http://geo.arc.nasa.gov/esdstaff/health/sensor/cfsensor.html`.
8. NASA, 2000. `http://landsat.gsfc.nasa.gov/`.
9. Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Communications* **COM-28**, pp. 84–95, 1980.
10. M. E. Leeser, J. Theiler, M. Estlick, N. V. Kitaryeva, and J. J. Szymanski, "Effect of data truncation in an implementation of pixel clustering on a custom computing machine," in *Reconfigurable Technology for Computing and Applications II*, *Proc. SPIE* **4212**, pp. 80–89, 2000.
11. Annapolis Microsystems, Inc., 2001. `http://www.annapmicro.com`.
12. Xilinx Corporation, 2000. `http://www.xilinx.com/products/virtex.htm`.
13. Synplicity Corporation, 2000. `http://www.synplicity.com`.
14. Xilinx Corporation, 2000. `http://www.xilinx.com/products/software/software.htm`.
15. Xilinx Corporation, 2001. `http://www.xilinx.com/products/logicore/coregen/index.htm`.

An Applications Approach to Evolvable Hardware

Reid Porter and Kevin McCabe
Los Alamos National Laboratory,
Space and Remote Sensing,
Los Alamos, New Mexico USA 87545.
rporter, kmccabe@lanl.gov

Neil Bergmann
Cooperative Research Centre for Satellite Systems ,
Queensland University of Technology,
Brisbane Australia 40001.
n.bergman@qut.edu.au

Abstract

We discuss the use of Field Programmable Gate Arrays (FPGAs) as hardware accelerators in genetic algorithm (GA) applications. The research is particularly focused on image processing optimization problems where fitness evaluation is computationally demanding and poorly suited to micro-processor systems. This research identifies key design principles for FPGA based GA and suggests a novel 2 stage reconfiguration technique. We demonstrate its effectiveness in obtaining significant speed-up; and illustrate the unique hardware GA design environment where representation is driven by a combination of hardware architecture and problem domain.

1. Introduction

Evolvable hardware attempts to apply evolutionary principles to the design of electronic circuits. We are also interested in using genetic algorithms to find suitable hardware circuits for particular applications but are motivated by the long execution times of software GA experiments. GAs are an effective optimization procedure which use a large number of candidate solutions (population) to converge over time to a global optimum [2]. Maintaining such a population leads to robust solutions in many problem domains but also leads to large computation times. This is particularly true when GAs are applied to image processing problems which are generally poorly suited to traditional sequential processors. Hardware acceleration of GAs is a difficult problem due to the application specific nature of representation, and fitness evaluation. FPGAs are a flexible implementation alternative that can provide the application specific architecture necessary for GA speed-up while still maintaining software flexibility.

We suggest FPGA based GA-accelerators using a novel 2-stage reconfiguration technique. A GA-accelerator needs to be configurable for (1) different problems and (2) different candidate solutions (also referred to as individuals) of the population for each problem. For each problem the FPGA is configured once at the start of a GA run using the FPGA programming bit-stream. This method is too slow to configure each individual of the GA population and so we introduce a second level of configurability using control lines to dictate arrangement of a fixed set of operators specific to that problem.

Section 2 introduces the two stage reconfiguration technique and associated terminology. The first and second levels of configurability are described in more detail in Sections 3 and 4 respectively. Section 5 describes the application of the technique to a multi-spectral feature identification problem and provides results from initial experiments.

The key to efficient implementation of the 2-stage reconfiguration technique is in the reuse of hardware resources from one individual to the next. This maximizes the amount of FPGA area being used at any one time leading to increased performance. We describe three levels of hardware reuse in this paper: gate level reuse is the most commonly used method in evolvable hardware systems and is described in Section 3, arithmetic level reuse is described in Section 4 and application specific operator level reuse in Section 5.1.

2. The GA Isostructure

Usually the most computationally intensive part of a GA is the fitness evaluation. This involves evaluating how well each individual in a GA population solves the particular problem. When GAs are applied to image processing problems this can involve several data intensive image processing operations for each individual. If each individual can be implemented in hardware, this fitness evaluation can be carried out at high speed, reducing GA run times considerably [5]. A problem with high density FPGAs is that reconfiguration time can often become large compared to the time required for fitness evaluation. To avoid this problem we suggest that two levels of configuration are required.

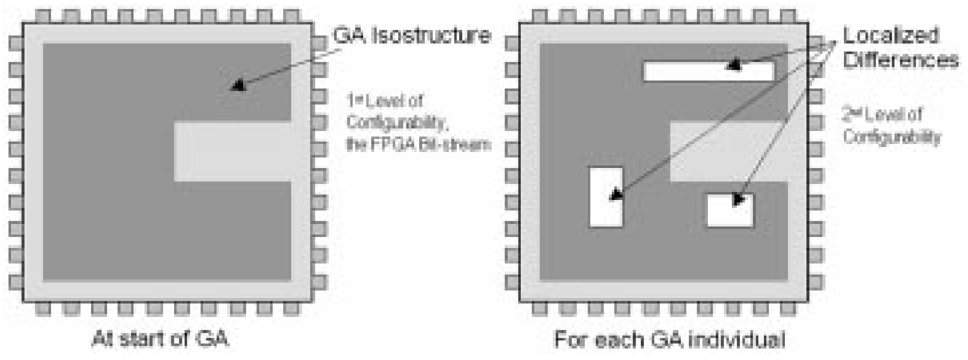

GA Isostructure
1st Level of Configurability, the FPGA Bit-stream
At start of GA
Localized Differences
2nd Level of Configurability
For each GA individual

Figure 1. Two stage reconfiguration

In most GA experiments, individuals have similar implementation requirements. A key to efficient implementation is to localize where GA individuals differ so that reconfiguration time can be minimized. We present Figure 1 and the following terminology for clarity. The GA isostructure is implemented with the first level of configurability, the FPGA programming bit-stream, and is common to all individuals in a GA run. The GA isostructure requires structure to obtain significant speed-up, and also flexibility to implement all GA individuals of interest. The second level of configurability is then used to rapidly fine tune the isostructure, by setting control lines, to implement particular GA individuals. The result of this two stage configuration process is a hardware implementation of a particular GA individual where fitness evaluation can be carried out at high speed.

The choice of isostructure is an important one and is dictated by a combination of application domain and hardware resources. It leads to unique hardware search spaces that will have direct effect upon the evolvability of the system. This will be discussed further in Section 5.2.

3. The First Level of Configurability

With rapidly reconfigurable FPGAs such as the Xilinx XC6000 series devices only the first level of configurability is required. FPGA programming is generally still a two step process. (1) An isostructure is first implemented at the start of the GA run and (2) particular GA individuals are then configured as required using partial reconfiguration. Since the GA is operating on the programming bit-stream directly, the functionality and connectivity of each FPGA logic cell can vary from one GA individual to the next. We consider this as hardware reuse at the gate level.

We investigated this feature by evolving cellular automata (CA) rule tables to perform basic pattern recognition. Papers by Mitchell [8] and Sahota [10] describe similar experiments performed in software. Our experiment, described in detail in [9], implemented 5 variable, 2 state CA on a XC6000 series FPGA using combinatorial logic trees. The logic tree isostructure is illustrated in Figure 2 and is configured by selecting inputs to the logic gates and multiplexers as well as choosing suitable functions for the logic gates. This isostructure made a high speed implementation possible while still providing flexibility to implement all GA individuals of interest, in this case a 5 variable CA rule table.

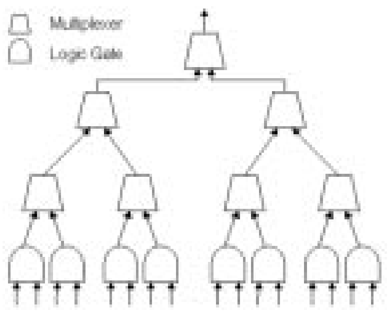


Figure 2. Logic Tree Isostructure

Representation based on the logic tree isostructure was compared against a software based CA rule table GA. It was found that the logic tree isostructure could implement problem specific constraints more easily than CA rule table representations leading to improved performance. Evaluating the fitness of the hardware individual was 38 times faster than software fitness evaluation.

4. A Second Level of Configurability

FPGA manufacturers have moved away from rapidly reconfigurable architectures in favor of high density devices. Such devices suffer from long reconfiguration times and therefore the second level of configurability is required. The FPGA bit-steam is suitable for configuring the GA isostructure since this is necessary only at the start of a GA run. The second level of configurability must be incorporated into the GA isostructure directly using multiplexers and control lines. Hardware reuse is an important design consideration at this stage since incorporating the control lines can quickly dominate hardware resources. Most applications do not require the fine grain flexibility of gate level configurability and therefore we adopt a problem orientated approach to hardware reuse.

Generalized pipelined arrays, suggested by Kamal in [4] are an excellent example of how the second level of configurability can be implemented. These arrays can be configured by setting control lines to perform a number of common operations such as multiplication, division, square root, and squaring. These operations have very similar hardware requirements and therefore can be combined with large area cost savings. An arithmetic array was implemented on Altera Flex10K devices and details are summarized in Table 1. Through intelligent reuse of hardware

Operator	$\times$	$\div$	$\surd$	$\times \div \surd$
Estimated Area Cost (Logic Cells)	206	286	305	429
Resource Gain				1.98

Table 1. Area Cost Estimates

the computational resources available to the GA is effectively doubled. We consider this as hardware reuse at the arithmetic level and suggest such arrays as useful building blocks for many GA applications.

5. Application to Multi-Spectral Feature Identification

Multi-spectral image processing is a data intensive and time consuming process traditionally employing groups of analysts to manually identify regions of interest within a particular data set. Automatic feature identification algorithms have been developed but are often application and data set specific and lack generality to be applied to the wide range of problems analysts may encounter. We attempt to use GAs to automatically generate and fine tune feature identification algorithms for particular applications or data sets that have not yet been considered. We are particularly interested in area-based features that are used in terrain classification and land use characterization.

Our approach is closely related to Genetic Programming methodologies where a number of image processing operators are combined with the multi-spectral input channels to produce executable algorithms. The choice of operators is an important one and is dictated largely by hand crafted classification algorithms developed in the remote sensing community.

5.1. Hardware Reuse at the Operator Level

Operators are grouped according to their hardware requirements and are summarized in Table 2.

Type	Examples
Spectral	NDVI, Linear Scale, Linear Combination
Spatial	Mean, Standard Deviation, Convolution, Rank Order
Threshold	Clipping, Boolean, Band Pass

Table 2. Types of Operators

Spectral operators are used extensively in the remote sensing community and are applied on a pixel by pixel basis to one or more input channels. Their data requirements suggest a pipelined architecture and we therefore use Kamal's generalized arithmetic pipeline as a starting point. A powerful measure of land type in remote sensing is the Normalized Differential Vegetation Index (NDVI) calculated as: $NDVI = \frac{A-B}{A+B}$ where A and B represent two multispectral input channels [6]. We use this problem specific knowledge to define the configurable spectral operator architecture of Figure 3.

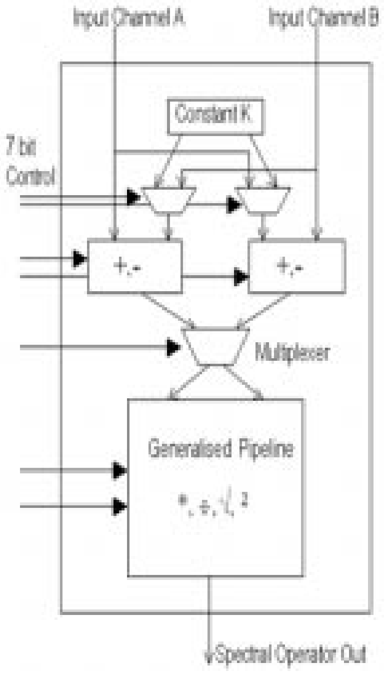


Figure 3. Programmable Spectral Operator

With the increasing spatial resolution of multi-spectral sensors, texture information becomes increasingly useful [3]. Spatial operators are applied to a local neighborhood of pixels taken from a single input channel. The predominate hardware cost for spatial operators in a pipelined architecture is accessing the local neighborhood and this can be shared between different operators. Thresholding operators also have very similar hardware requirements and can share resources easily. By combining operators into larger, programmable macro operators we can employ hardware reuse at the operator level.

5.2. Choice of Isostructure

A GA individual will contain a number of operators in a specific order. One individual may use a spectral operator followed by a spatial operator, while another will use a spatial operator followed by a spectral operator. In software the GA can be used to find the optimum order of operators. In hardware some structure must be imposed on the search space in order to make implementation possible. The effect of GA isostructure on evolvability is not clear and knowledge of the problem domain plays an important role. Miller in [7] discusses similar issues of evolvability in evolving 2-bit binary multipliers. Adaptive representation schemes have been suggested [1], which may be applicable to this problem. Since isostructural optimisation would only need to be performed once at the design stage, efficiency gains may be possible if the problem domain has general structural properties.

5.3. Summary of Initial Experiments

The isostructure used in our initial experiments is illustrated in Figure 4. The GA selects the input channels to the top level spectral and spatial operators, as well as the precise functionality of the 5 image operators. Figure 5 is an example of the training data used showing a visible band of the multi-spectral input (top) and desired output image (bottom). The required area feature, in this case a type of vegetation, was identified by hand and used as binary truth.

Experiments used a population of 100 and ran for 30 generations using an elitest strategy similar to that used in [8]. Crossover and mutation were constrained to maintain the GA isostructure. We compared the performance over 30 runs of the structured GA to a software version where operator order was not specified. Both types of GA performed equally well with an isostructural individual achieving the highest overall accuracy of 95.7%.

There was not enough statistical difference in the results to discern the better GA strategy, but we are encouraged by the similar performance and conclude the GA isostructure has the necessary flexibility to implement most individuals of interest. Further investigations are required including

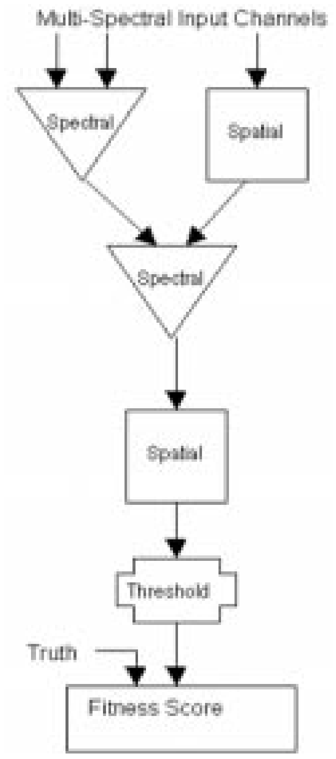


Figure 4. GA isostructure

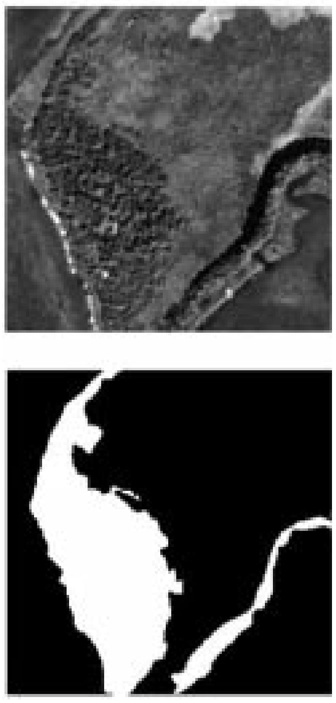


Figure 5. Example training images

Operator	#Logic Cells (12160 Total)	% of FPGA
Spectral	650 * 2	11
Spatial	1230 * 2	20
Threshold	125	1
Total	3885	32
On Chip Memory	8Kbits out of 40Kbits	20

Table 3. Area Cost Estimates

variation of the GA isostructure. We believe an isostructural GA is possible that will include all GA individuals of interest and may even out-perform unconstrained GA runs due to the reduced search space.

We have estimated the FPGA area requirements for the GA isostructure of Figure 4 which are summarized in Table 3. The design is targeted at an Altera Flex10K250 device and will be implemented in the near future. The isostructure uses one third of the FPGA resources indicating more complicated structures are possible. Future experiments will look at making use of these resources as we learn more about the problem with new operators, advanced classification techniques and intelligent fitness case selection. The target clock speed of the GA isostructure is 40Mhz which achieves a speed-up of 50-100 times an optimized Pentium-II, 450 MHz implementation.

6. Conclusion

Some key concepts in the implementation of FPGA based GA-accelerators have been identified. These include a two stage approach to reconfiguration made necessary by long reconfiguration times of high density FPGA devices. The first level of configurability, the FPGA bit-stream, is used to implement a structure common to all GA individuals at the start of a GA run. This structure is referred to as the GA isostructure and is fine tuned with the second level of configurability to implement all GA individuals of interest. This second level of configurability must be incorporated into the GA isostructure directly by using control lines and multiplexers.

We identified the importance of hardware reuse in implementing the second level of configurability and suggested a problem orientated approach. Reuse at several levels was discussed and arithmetic building blocks suggested. We discussed the importance of GA isostructure, dictated by a combination of problem domain and hardware resources, on evolvability and described initial experiments in multi-spectral feature identification.

References

[1] L. Altenberg. Evolving better representations through selective genome growth. *Proceedings 1st I.E.E.E. Conference on Evolutionary Computation*, June 1994.

[2] D. Golberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[3] R. Haralick and K. Shanmugam. Combined spectral and spatial processing of erts imagery data. *Remote Sensing of Environment*, 3:3–13, 1974.

[4] A. K. Kamal, H. Singh, and D. Agrawal. A generalized pipeline array. *I.E.E.E. Transactions on Computers*, C-23:533–536, May 1974.

[5] J. R. Koza et. al. Rapid reconfigurable field-programmable gate arrays for accelerating fitness evaluation in genetic programming. *Late Breaking Papers at the Genetic Programming 1997 Conference*, pages 121–131, 1997.

[6] C. Leprieur, M. Verstraete, and B. Pinty. Evaluation of the performance of various vergetation indices to retrieve vegetation cover from avhrr data. *Remote Sensing Reviews*, 10:265–284, 1994.

[7] J. Miller and P. Thomsom. Aspects of digital evolution: Evolvability and architecture. *Parallel Problem Solving from Nature - PPSN V. 5th International Conference*, September 1998.

[8] M. Mitchell, J. P. Crutchfield, and R. Das. Evolving cellular automata with genetic algorithms: A review of recent work. *First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*, 1996.

[9] R. Porter and N. Bergmann. Evolving fpga based cellular automata. *SEAL'98 : Simulated Evolution and Learning*, October 1998.

[10] P. Sahota, M. F. Daemi, and D. G. Elliman. Training genetically evolving cellular automata for image processing. *International Symposium on Speech, Image Processing and Neural Networks*, April, 1994.

An Application-Specific Compiler for High-Speed Binary Image Morphology [*]

Scott Hemmert, Brad Hutchings
Brigham Young University
{hemmert, hutch}@ee.byu.edu

Abstract

This paper discusses a two-level compilation scheme used for generating high-speed binary image morphology pipelines. The first-level compiler generates a generic morphology machine which is customized for a particular set of instructions by the second-level compiler. Because the generic machine is re-used instead of re-synthesized every time, we are able to acheive compile times similar to software compile times, while still acheiving a 10X speed-up over the software implementation.

1 Introduction

The Focus of Attention (FOA) algorithms are used by Sandia National Laboratories as a preprocessing stage to their Automatic Target Recognition engine. FOA is used to process synthetic aperature radar (SAR) images with the goal of finding regions in the image which contain likely tagets. The exact FOA algorithm used is dependent on the type and size of targets as well as the radar system generating the images. Weather conditions can also cause changes to be made to the algorithm. The goal of FOA is to reduce the size of the data set for the more computationally complex stages of ATR which identify the existence and type of targets.

Because of its demanding computational requirements, FOA was originally implemented on a specialized morphology computer, called the CYTO computer [1], manufactured by the Environmental Research Institute of Michigan (ERIM). The CYTO computer was constructed using custom ASICs to create a semi-programmable, image-processing pipeline computer. The CYTO computer could be programmed to perform a wide variety of image morphology operations using an arcane language known as C4PL (CYTO Portable Parallel Picture Processing Language)[2]. An FOA script then is implemented as a list of C4PL operators performed in sequence. Due to improvements in IC fabrication, the CYTO computer quickly became obsolete and has been replaced by a software implementation. However, C4PL outlived the CYTO computer and is still in use as the programming (or scripting) language used to describe FOA applications.

The current software implementation by Sandia National Laboratories is a collection of optimized C functions which emulate the operation of a subset of C4PL instructions. This software reads the FOA script and chains together the operations specified in the script. Due to advances in microprocessors, this sequential software approach is many times faster than the original CYTO computer.

Even though the software version of FOA was a vast improvement over the CYTO computer, it still did not reach the performance goals set forth by Sandia, and accelerating FOA with custom ASICs, as was done with the CYTO computer, is not an option because the final solution must use COTS (Commercial off-the-shelf) hardware. In addition to performance requirements, Sandia also put some requirements on programmability and programmer usability:

- Assume programmers have no hardware experience. Image Processing specialists write the algorithm.
- Actual FOA algorithms used are classified. The algorithm must be specified and mapped to hardware without involving unclassified personnel.
- The algorithms are to be specified in the C4PL programming language.
- Because operating conditions may dictate an algorithm change, in-field modifications must be possible. The maximum compile time that can be tolerated is half a work day (4 hours).
- Final solution must be embeddable in an approved ruggedized form factor.

To meet these requirements, we mapped the design to Xilinx XC4000 family FPGAs and utilized a two-level compilation approach. The first stage of the compiler generates a structural design specific to a particular platform, which provides the generic hardware structure sufficiently large to perform the desired operations. The second phase

[*]Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

customizes the generic pipeline to perform the specified C4PL operations. This paper discusses the evolutionary steps along the path leading to the final approach and implementation and compares the various approaches both for circuit size and compilation speed. We will also compare our final hardware approach with Sandia's software implementation.

2 Background

The C4PL instructions have their roots in binary image morphology. In this section, we will discuss some of the basics of binary image morphology as well as the general structure of the relevant C4PL instructions.

2.1 Binary Morphology

Binary morphology [1] consists of a set of operations used to study and change geometric properties in binary images. It can be used to find, enhance and/or remove certain geometric features in an image. For example, it can be used to look for corners, close small gaps in an object, etc. The most important operations for our purposes are dilation, erosion and the hit-and-miss transform. All of these operations are based on operations on two or more sets. One of these sets is the image on which we are operating and the members of the set represent the *on* pixels in the image; the other sets are refered to as structuring elements. In general, a structuring element can be any size and have its origin at any location within its bounds. A possible 3x3 structuring element is shown in Figure 1.

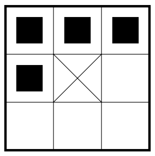


Figure 1: Possible structuring element used in binary morphology. The X indicates the origin of the image, and black squares indicate *on* pixels.

Elements in the sets are represented by ordered pairs in a two dimensional Euclidean space (E^2). An ordered pair indicates the position relative to the origin of the image. For example, the structuring element in Figure 1 would be represented by the set {(-1,0),(-1,1),(0,1),(1,1)}.

[1]Morphology is derived from `morph` meaning shape and `-ology` meaning the study of. Thus, morphology means the study of shape.

The following sections will briefly describe the purpose and operation of the three morphological operations mentioned above. For a more detailed discussion of binary morphology, please refer to [3] or [4].

2.1.1 Dilation

Dilation, denoted by the operator $\oplus$, combines the image set with the structuring element set using Minkowski addition [2], pairwise on all elements of one set with all elements of the other. This is described mathematically as:

$$A \oplus B = \{c \in E^2 : c = a + b, a \in A \text{ and } b \in B\}$$

As an example, consider the following sets:

$$\begin{aligned} A &= \{(1,3),(1,2),(2,2),(2,1)\} \\ B &= \{(-1,0),(0,0),(1,0)\} \\ A \oplus B &= \{(0,3),(0,2),(1,2),(1,1),(1,3), \\ &\quad (2,2).(2,1),2,3),(3,2),(3,1)\} \end{aligned}$$

This is illustrated graphically in Figure 2(a).

Dilation is generally used to fill small gaps in images, or to grow objects. It can also be used in conjunction with other operations to remove noise, etc.

2.1.2 Erosion

Erosion, denoted by $\ominus$, is the dual of dilation. It combines the image and structuring element sets using a subtraction-like operation. The operation is described mathematically as:

$$A \ominus B = \{c \in E^2 : c + b \in X \text{ for every } b \in B\}$$

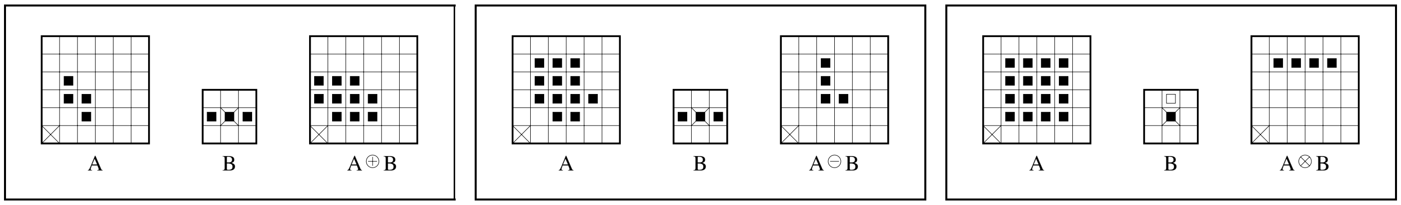
In practice, this means that the structuring element is compared against every pixel in the image. If all *on* pixels in the structuring element are also *on* in the image, then the pixel under the origin of the structuring element is *on* in the output image.

An example of erosion is given in Figure 2(b).

2.1.3 Hit-and-Miss

The hit-and-miss transform is used to select pixels that have particular geometic properties. The hit-and-miss transform can be computed many different ways. The most intuitive way to think of this transform is to look at it as a specialized template match. Along with checking whether some points belong to the image set, we can also check to make sure that

[2]Minkowski addition is an element wise addition. For example $(1,2)+(3,4) = (1+3, 2+4) = (4,6)$.



(a) Dilation

(b) Erosion

(c) Hit and Miss

Figure 2: Binary Morphology Examples. Dark squares denote *on* pixels. The white square in the hit-and-miss template denotes a pixel which must match an *off* pixel.

other points do not belong to that set. The template is specified by two sets, B_1 and B_2. The B_1 set specifies positions which must match *on* pixels and the B_2 set specifies positions which must match *off* pixels. The output set contains all those points which match both the B_1 template with *on* pixels and the B_2 template with *off* pixels. We descibe this mathematically as follows:

$$A \otimes (B_1, B_2) = \{x : B_1 \subset A \text{ and } B_2 \subset A^c\}$$

Since the elements of the B_1 and B_2 sets are required to be mutually exclusive, it is possible to specify the set of templates as a single template where each location has one of three possible values: *on*, *off* and *don't care*. Pixels in the B_1 set are marked as *on*, and pixels in the B_2 set are marked as *off*. All others are *don't care*. We call this combined template B.

An example of a hit-and-miss transform is given in Figure 2(c). The template in this example picks out pixels along upper edges.

2.2 C4PL Instructions

As with morphological operations, C4PL instructions aim to modify or enhance certain geometric properties. The C4PL instructions are both a generalization and a narrowing of binary image morphology. They are more narrow than binary morphology in that all C4PL instructions use only 3x3 templates with the origin in the center. This limits the size of patterns that can be matched, but still leaves enough flexibility for the problem at hand.

However, C4PL instructions are much more general in that an image may be seperated into more than two sets. In binary morphology, an image is divided into two sets, an *on* set and an *off* set; CYTO allows the image to be split into multiple sets. This means that each pixel in an image is assigned to one of many possible sets. In this context, we refer to these different sets as planes or states. For example, referring to plane 3 would refer to all pixels in the image which have been assigned to the set labeled 3.

There are two main ways to represent the assignment of pixels to the different planes. The first and most straightforward approach, called pixel-level encoding, is to use a single multi-bit value for each pixel. The value representing the pixel would be set to the plane number to which that pixel has been assigned. This method was used by the ERIM CYTO Computer, and has the advantage of being compact, but the disadvantage of requiring the instructions to decode the pixel to determine if that pixel is or is not in a particular set.

The second approach, refered to as plane-level encoding, is used by the Sandia software implementation and uses multiple mutually exclusive single-bit (binary) images. Each binary image represents one of the possible sets to which the pixels can be assigned, and each pixel in the image has a corresponding pixel in each of these binary images. While this approach takes more memory to represent a given image, it is more computationally efficient for software, as each plane can be packed into the native word size of the microprocessor, thus allowing the microprocessor to operate on multiple pixels at a time.

The FOA algorithms use only a small subset of the available C4PL instructions. The operations used in FOA scripts as well as a brief description of their functions are as follows:

- span: Dilation operation which uses a 3x3 structuring element filled with 1's.
- spandisk: Dilation operation which alternates using an 8-way connected neighborhood and a 4-way connected neighborhood as its structuring element.
- tranb: Dilation or erosion using user specified structuring elements.
- skelrec8: Skeletonization operation.
- cover: Moves pixels from one set into another.
- exch: Exchanges two sets.

The above instructions can be grouped into two different types of operations: neighborhood and scalar. Neighborhood operations look at a pixel as well as its eight neighbors and act dependent on some template criteria in the neighborhood being met. Scalar operations act on a single pixel at a time and are unconditional.

Since a general knowledge of the structure of these operations is necessary to understand the motivation for our implementation, we will show the general form of each type of instruction and give a detailed description of one instruction of each type.

2.2.1 Neighborhood Operations

We will first look at a simple example of a neighborhood operation, then we will look at the general form of this class of instructions. The *span* instruction is a conditional dilation and has the following form:

$$span\ \ f\ \ m\ \ o\ \ iter$$

The span operation simply passes through a pixel unchanged unless the following conditions are met: First, the pixel belongs to plane m, and second, at least one of the pixel's neighbors is in plane f. If these conditions are met, then the pixel is moved to plane o. This operation is repeated $iter$ times.

All neighborhood operations are similar to the *span* operation, in that they simply pass a pixel through unchanged unless a certain set of criteria is met. This criteria is very similar to that used to determine if a pixel is turned *on* in the output of a hit-and-miss transform. However, where the hit-and-miss transform uses a single template, a neighborhood operation can use any number of templates. The neighborhood is said to match if any of the templates matches. Not only must one of the templates match, but the pixel of interest must also be in the specified plane. Thus, a neighborhood instruction only effects pixels in a single plane.

2.2.2 Scalar operations

The scalar operations in the portion of C4PL we implement are very simple operations. These operations depend only upon the value of a single pixel and can typically be implemented with a single gate. For example, the instruction $cover\ \ i\ \ o$ would move any pixel in plane i into plane o. This is essentially the *or* of these two planes.

3 General Approach

Our general approach to the problem was to create a compiler that would generate a highly parallel, deeply pipelined circuit which implements a specified FOA script.

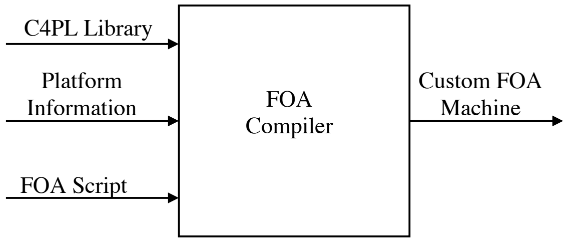


Figure 3: General approach for the FOA compiler.

This approach is described graphically in Figure 3. The compiler takes as input a C4PL instruction library, information on the platform to be compiled to and the FOA script of interest. The road to our final solution led through three evolutionary steps. The compilers progress from generating highly optimized circuits with slow compile times to creating a more generic circuit with extremely fast compile times. Each of these steps uses a slightly different form of the three inputs.

The circuits generated by the compiler are capable of accepting and processing a new pixel every clock cycle. This is done by streaming the image into the circuit in raster order starting in the upper left hand corner of the image and proceeding line by line. Each of the circuits has the same general structure. Each one uses delay lines to generate a 3x3 neighborhood. The contents of this neighborhood are passed to some calculation logic which actually computes the result of the instruction. This structure is show in Figure 4.

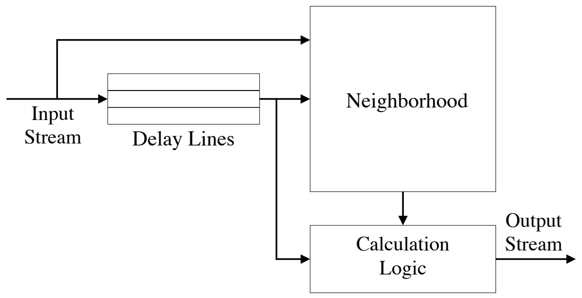


Figure 4: General structure of the FOA circuits.

A consequence of using delay lines to generate the neighborhoods is the need to "pad" the image. Padding the image consists of marking the end of each line so that the hardware knows where one line ends and a new line begins.

The following section will discuss the two types of cicuits we used in our compiler implementations. The next section will then discuss the three FOA compilers in greater detail.

4 FOA Circuits

We took two separate approaches to designing FOA circuits. The first approach was to create an optimized circuit for each C4PL instruction; the second approach uses a more generic circuit which has all the circuitry needed for all instructions and is easily specialized for specific instructions. The following sections will discuss the general structure of these two approaches. Specific information about a circuit implementation will be discussed with the compiler(s) which use the circuit.

4.1 Instruction Specific Circuits

The instruction specific circuits consist of a different circuit description for each C4PL instruction used in writing FOA scripts, both neighborhood and scalar operations. The scalar operations can be implemented in a single gate or less. The neighborhood operations are all similar and have the general structure shown in Figure 4. The main difference in each instruction is the logic found in the box labeled calculation logic. Each instruction uses the minimum amount of logic to implement the desired function. In this way, a given FOA script can be implemented in the smallest circuit area possible.

4.2 Generic C4PL Circuit

The main feature of this type of circuit is that all instructions are represented by a single circuit with configurable elements which are used to implement specific instructions. In order to create a Generic C4PL Instruction (which we call a *Supercell*), it was necessary to identify the common features in all the instructions used to write FOA scripts. As was discussed in Section 2.2.1, all neighborhood instructions must do the following:

1. generate an 8-way connected neighborhood,
2. retreive the value of the center pixel of the neighborhood,
3. match any number of templates based on the generated neighborhood, and
4. calculate the output of the instruction based on the output of the template matchers and the value of the center pixel.

Of these four functions, the first two are identical for every neighborhood instruction. In order to change the instruction, only the structure of the last two must change.

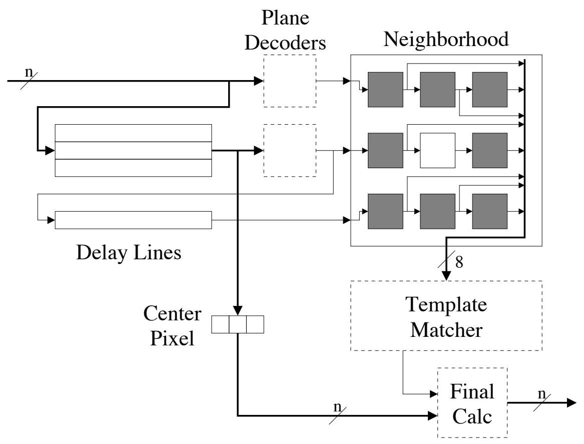


Figure 5: Block diagram of generic C4PL instruction (SuperCell). Bold lines represent multi-bit signals and regular lines represent single bit signals. Elements drawn with dashed lines are the reconfigurable elements which are modified to create specific instructions.

Since this circuit uses pixel-level encoding, decoders are needed at the input of the neighborhood generation circuitry in order to pick out pixels from the proper plane. The following sections will describe the six main features of a Supercell as seen in Figure 5. The circuitry can be split up into two main types: fixed circuitry and programmable circuitry.

Fixed Circuitry The majority of the circuit is made up of elements which are identical for all C4PL neighborhood instructions, and therefore need not change in order to implement different instructions. The following are brief descriptions of this circuitry:

- **Delay Lines.** The delay lines are generated using RAM16x1s (16x1 synchronous RAM) primatives and a linear feedback shift-register (LFSR)[5].
- **Neighborhood.** In conjunction with the delay lines, the neighborhood circuitry builds the 8-way connected neighborhood for a pixel. The circuit is built from 9 flip-flops.
- **Center Pixel.** This is simply a register used to delay the center pixel two extra cycles so that it is synchronized with the center pixel in the neighborhood generator.

Programmable Circuitry The programmable circuitry is the heart of this implementation, and allows the generic circuit to implement any neighborhood operation. The circuit

Unit	Lookup Table Size		Number of Bits in the Table			
	Input Bits	Output Bits	n = bits per pixel	$n = 3$	$n = 4$	$n = 5$
Template Matcher	8	1	256	256	256	256
Plane Decoder	n	1	2^n	8	16	32
Final Calculation	$n + 1$	n	$n(2^{n+1})$	48	128	320

Table 1: Details of the input and output width of the lookup tables used to implement the programmable circuitry, as well as the number of programming bits needed to program each unit of a Generic C4PL instruction. This number corresponds directly to the size and number of RAMs needed to implement the necessary logic. n represents the number of bits used to represent a pixel.

uses lookup tables to generate all the programmable circuitry. The generic instruction is made into a specific instruction by modifying the contents of these tables.

As all the programmable elements are built from lookup tables, it is interesting to look at the number of bits needed to implements each unit. This number determines the size of the lookup table as well the number of bits needed to reprogram the generic instructions. Table 1 shows the size of the lookup table required for each element. The following sections will briefly discuss the purpose of each of the programmable circuit elements.

- **Plane Decoders.** The plane decoders convert the multi-bit pixel to a single bit which indicates whether or not the pixel was a member of the designated neighborhood plane.
- **Template Matcher.** The template matcher takes the eight bits surrounding the center pixel and determins if they match any of the templates given in the instruction definition.
- **Final Calculation.** The final calculation unit looks at the output of the template matcher and the value of the center pixel and decides what the output should be.

4.2.1 Scalar Folding.

Since the final calculation is implemented as a lookup table, it is possible to use it to "fold" in any number of scalar operations which follow it. This is easy to see if you consider that any scalar operation can be implemented as an n input n output lookup table (where n is again the number of bits used to represent a pixel). This allows any plane to be mapped to any other plane. The new lookup table is simply the cascaded result of the original final calculation unit followed by the lookup tables associated with each scalar operation.

4.2.2 Control Characters

An advantage of using pixel-level encoding is that we are able to use some of the pixel values as control characters. We need only two control characters: valid character and edge character. The valid character is used to tag the beginning and end of an image. This information is needed by the back-end circuitry so that it can properly write the processed image to memory. The edge character is used to pad the image as discussed in Section 3. We use the two highest possible pixel values for these signals. These values always get passed through instructions unchanged. Further, plane decoders always output zero for either of these values. This ensures that these characters will not effect the processing of the image.

4.2.3 Pixel Size

This implementation allows us to change the number of bits used to represent a pixel. Chosing the size requires striking a balance between the number of usable planes and the size of the circuit. Using more bits to represent a pixel gives the FOA programmer a larger number of planes to use, but also greatly increases the size of the circuitry. From discussions with Sandia, we determined that 3 bits per pixel would provide enough available planes for the FOA programmer. So, all of our *Supercell* implementations use 3 bits per pixel.

5 FOA Compilers

As previously discussed, we went through three steps to reach our final implementation. The first approach attempted to make the most efficient use of circuit area and therefore used custom instruction specific circuits. This first approach met the desired performance goals, but did not meet the required compile times, so the next two approaches aimed at speeding up the compile time. These approaches used the *Supercell* circuit described in Section 4.2, and introduced a two-level compilation scheme which would dramatically reduce the in-field compilation time.

The following sections will discuss the three compiler approaches in greater detail. In addition to providing a description of the compilers, we will also compare the implementations with repect to the following areas:

1. Average size (in Xilinx XC4000 CLBs) for a neighborhood operation. The neighborhood size is dependent on the width of the image to be processed. Typically, the circuit would process a 1024 pixel width image. However, because of limited resourses on the board, our circuitry was build to handle images with a width of 576 pixels. The image is then broken up and processed seperately.
2. Compiler speed.

We will also compare the throughput of our final approach with Sandia's software implementation.

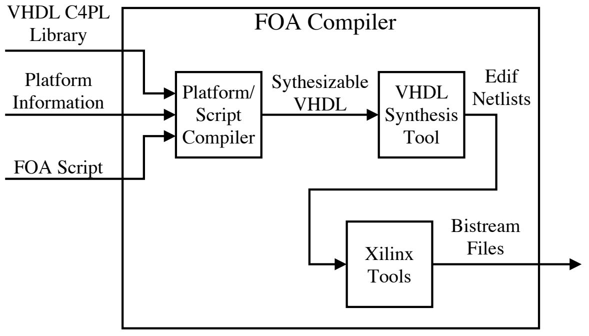


Figure 6: Structure of Direct Synthesis Compiler.

5.1 Direct Synthesis

Our first compiler generated a new custom machine for each FOA script using instruction specific circuitry. The C4PL library for this compiler consisted of synthesizable, parameterized VHDL models for each C4PL instruction of interest. The library also includes the circuits which read image data from memory and write the processed image to memory. The platform information was built directly into the compiler and targeted the Annapolis Microsystems Wildforce board. This approach can be seen in Figure 6. The output of the compiler was bitstreams which contained designs implementing the input FOA script. To properly create these bitstreams, the compiler was reponsible for completing the following:

- **Instruction sequencing.** The main responsibility of the compiler was to instance library elements in the order specified in the FOA script.
- **Chip-level partitioning.** Typical FOA scripts were too large to fit in a single FPGA, so the compiler had to determine how many instructions would fit in an FPGA, and do chip-level partitioning of the design. The compiler would output multiple bitstreams, each one corresponding to a single FPGA.
- **Unused plane synchronization.** Since this implementation used plane-level encoding and planes used in an instruction incur a latency equal to the width of the image being processed plus 2, planes not used in instructions must be delayed this same number of cycles so that the planes remain in synchronization.

Because the number of planes used is dependent on the FOA script, the actual size of the circuit is also script dependent. Typical FOA scripts generated average neighborhood operation sizes of 81 CLBs. Because of the long tool flow, the design did not meet the constraints given us for fast compile times; average compile times were on the order of 10 hours for a typical script; 99+% of that time was spent in the synthesis tool and Xilinx tools.

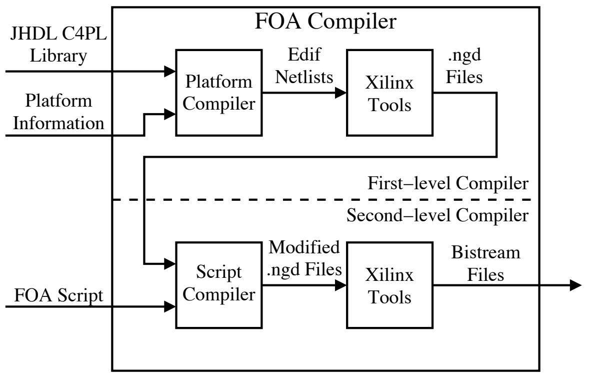


Figure 7: Structure of the Generic FOA Machine compiler.

5.2 Generic FOA Machine

The second approach concentrated on reducing the compile time. At this point, we introduced a two-level compilation scheme as shown in Figure 7. The first stage generates a platform specific, script generic FOA circuit. The generic FOA circuit is made up of a linear array of *Supercells* as discussed in Section 4.2. The second stage uses the input FOA script to customize the generic machine, by modifying the contents of the lookup tables. The following sections will describe these two compilation stages in greater detail.

5.2.1 First-level Compiler

The first-level compiler used as input a JHDL [6, 7] module library and targetted the Annapolis Microsystems Wildforce board. The module library included a *Supercell* module generator as well as circuitry to convert image data to/from the image streams needed by the circuit. The *Supercells* generated by this compiler use simple ROM cells to implement the lookup tables used for reconfiguration.

The responsibility of the first-level compiler is to generate a generic FOA machine built from *Supercells* which utilizes all the available space on the Wildforce board, given the size of parts available on the board. The output of this stage is placed and routed `.ngd`[3] files. To generate the `.ngd` file, the Platform Compiler (see Figure 7) creates an edif netlist for each FPGA on the Wildforce board. These netlists are then passed through the Xilinx tools which generate the `.ngd` file.

This compilation stage takes about 10 hours to complete. However, this stage need only be done once for a given Wildforce board. The `.ngd` files created by this stage are stored and used as input to the next compilation stage.

5.2.2 Second-level Compiler

The inputs to the second-level compiler are the `.ngd` files generated by the first level compiler and the FOA script to be implemented. In order to customize the generic machine created in the first stage, the second-level compiler must complete the following:

- **Determine Programming Bits.** For each neighborhood instruction in the FOA script, the compiler determines the contents of the lookup tables for the plane decoders, template matcher and final calculation unit, which will implement that instruction.

- **Scalar Folding.** Each scalar operation is folded into the final calculation unit of the neighborhood operation preceeding it. If there are multiple consecutive scalar operations, they are folded in one at a time.

- **Modify `.ngd` Files.** The script compiler (as seen in Figure 7) parses the `.ngd` files looking for the proper ROM cells to modify. As it finds each ROM, it modifies its contents to reflect the operations specified in the FOA script.

- **Generate Bitstreams.** After the script compiler modified the `.ngd` files, they are passed through the remaining Xilinx tools to generate bitstreams.

This compilation stage takes, on average, 30 minutes to complete, about 95% of this time is spent in the Xilinx tools. Although this method was much faster than our previous approach, it has one large disadvantage: the `.ngd` file is a proprietary Xilinx file, and we have no guarentee that the format will not be changed in the future. If this file format was changed, the second-level compiler may no longer work correctly.

[3]A `.ngd` file is a Xilinx proprietary file which represents mapped, placed and/or routed designs targetting Xilinx FPGAs.

5.2.3 Compiler Speed and Circuit Size

As only the second stage was needed to customize a circuit in the field, this approach greatly decreased our "in-field" compile time; however, to accomplish this, we had to sacrifice some circuit size. Average compile time for this approach was about 30 minutes, a 20 times decrease over our previous approach. However, the size of a neighborhood operation grew to 88 CLBs, a 8.6% increase over the direct synthesis approach. Although the compile times were well within the maximum dictated by Sandia, the uncertainty of future compatability with the `.ngd` file format made us look at other options.

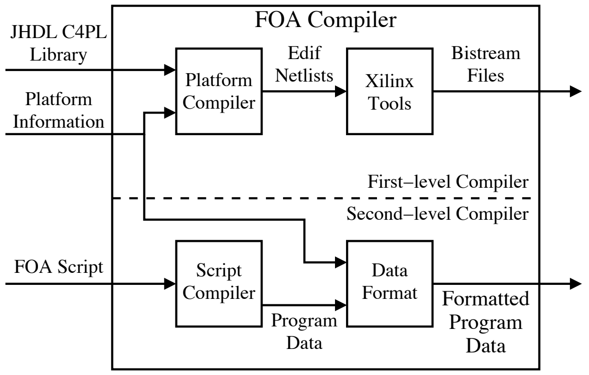


Figure 8: Structure of the Generic FOA Machine with Runtime Reconfiguration compiler.

5.3 Generic FOA Machine with Runtime Reconfiguration

Our final approach is similar to that disussed in Section 5.2. The main difference is that the ROMs containing the computation logic of the *Supercell* were converted to RAM cells so that the contents could be changed at runtime. This simple change allowed us to move from device-level configuration to user-level configuration. This meant that we could use fine-grain logic reconfiguration to change the function of our circuit at runtime.

The first-level compiler creates bitstreams which implement the generic FOA machine. The second-level compiler generates the programming bits needed to program this machine for a specific script. This compiler structure can be seen in Figure 8. The remainder of this section will discuss each part of this two-level compilation scheme in greater detail.

5.3.1 First-level Compiler

The first-level compiler is responsible for creating the generic framework for a specific hardware platform. The compiler originally targetted the Wildforce board, as did the other two compilers, but was later abstracted so that it might support other boards as well, although we did not retarget the compiler to any other XC4000 based configurable computing boards. The C4PL library for this compiler was written in JHDL and included the *Supercell* module generator, as well as the circuits used to convert image data to/from a raster stream readable by the FOA circuit.

The *Supercell* used by this compiler was slightly different than that used in the previous compiler. There are two major differences:

1. **Runtime Programmable Lookup Tables.** This version of *Supercell* used RAM cells instrumented such that they could operate in two modes: programming mode and lookup table mode. This was accomplished by inserting circuitry as shown in Figure 9.

2. **Cell ID.** Each *Supercell* is assigned a unique ID so that it is possible for the hardware to determine which programming bits belong in which *Supercell*. Programming bits will be ignored by all *Supercells* except the one which matches the specified ID. For more information, see Section 5.3.2.

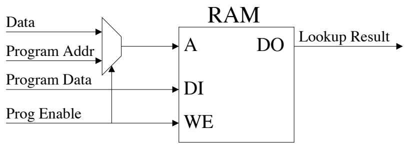


Figure 9: RAM cell instrumented to be a runtime programmable look-up table. Program mode is entered by asserting the Prog Enable signal. This will assert the write enable signal of the RAM as well as force the mux to pass the Program Addr signal to the address pins of the RAM.

The output of the first-level compiler is a set of bitstreams which implement a generic FOA machine capable of reading programming bits from memory and using this data to reprogramming itself for a specific FOA script. This process can take several hours to complete, but again, it need only be done once for each platform.

5.3.2 Second-Level Compiler

The second-level compiler is responsible for computing the programming bits needed to represent a given FOA script. It is also responsible for generating any control signals needed for programming. The second level compiler is accomplished in two stages. The first stage is platform independent and is responsible for the following:

- **Determine Programming Bits.** The compiler parses the FOA script and determines the contents of the lookup tables in each *Supercell*.

- **Scalar Folding.** The compiler folds scalar operations into the final calculation unit of the neighborhood operation preceeding it.

- **Generate Intermediate Format.** The first stage outputs an intermediate format, either as a file or a data structure, which simply contains the contents of the lookup tables that would implement the given FOA script.

The second stage takes the intermediate format produced by the first stage and information on the target platform and then formats the programming data, adding extra control/addressing data as necessary. This output is loaded into a memory on the board and the circuit uses it to reprogram itself. For the Wildforce board, we used a programming word as shown in Figure 10. Each control word specifies which cell (by ID number), element (plane decoder, template matcher, or final calculation) and address within the lookup table the data is intended for.

Both stages of the second-level compiler are completed in 1-2 seconds, depending on the size of the script.

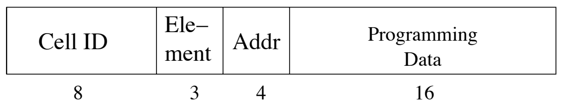


Figure 10: Programming format for the Wildforce implementation of FOA. The numbers represent the number of bits in each field.

5.3.3 Circuit Size and Compiler Speed

Because of the extra circuitry needed for the runtime reprogramming of the instruction pipeline, the size of a neighborhood operation grew 19.3% over the size for the last approach to 105 CLBs. However, by sacrificing some circuit area we were able to reach compile times of 1-2 seconds. This is a 900 times improvement over the previous two-level compilation scheme.

This improvement in compiler speed was made possible by the move away from device-level configuration to user-level configuration. This change was important because of the proprietary format of the `.ngd` file and other intermediate files. Because we cannot easily generate or modify a design at the bitstream level, device-level configuration requires us to use vendor tools to generate the bitstreams. Even if it were possible to modify the bitstream directly, we would need to map all the logical elements in the circuit to physical locations in the bitstream.

For our purposes, user-level configuration also gives us another advantage: With user-level configuration, the vendor tools are not needed to reprogram the machine for a specific script, and it may not be easy to make the vendor tools readily available once the platform is deployed. User-level configuration in this manner is only feasible for the application, because we did not need to change any of the routing in the circuit. Since the routing programming bits are not user accessible at runtime, any routing changes would need to be implemented with tri-state or muxes, which would require extra circuitry.

Moving to user-level configuration also allowed us to reduce the size of the data used to represent the essential data of an FOA script. In the first approach, the script was represented by the complete set of bitstreams used to implement the script. This means that we required 7,169,020 bits to represent a script [4]. The subsequent approaches represented the pertinent information as the contents of the lookup tables. Each Supercell can be reprogrammed using 312 bits. An average script uses about 125 Supercells; this would correspond to 39,000 bits. This is only 0.5% of the size needed to represent a script with the first approach.

5.3.4 Hardware vs Software Implementation

All of our approaches ran at about the same clock frequency. The last approach ran slightly slower because of the muxes needed on the input to all the programmable cells. The Xilinx tools reported that our final implementation would operate at 50 MHz. However, the memories on the Wildforce board we were using were limited to operating at 46 MHz, so the circuit was never tested faster than this.

Because the circuit is pipelined and capable of processing a new pixel every clock cycle, the 46 MHz circuit would have a peak throughput of 46 MegaPixel per second. Average throughput, taking into account the control overhead is about 44 MegaPixel per second. This throughput is the same no matter what FOA script is used. The software written by Sandia does not exhibit this same behavior. Because of the inherently sequential nature of the software implementation, the throughput is script dependent. On average, we measured the software implementation to get 4.3 MegaPixel per second on typical length scripts using a G4 PowerPC operating at 450 MHz. We chose to compare to this processor, as it is easily embeddable. For comparison, Table 2 includes numbers for other common microprocessors. This means that the hardware implementation has a 10X performance increase over the software implementation, for average size scripts.

[4]An XC4062 bitstream contains 1,433,804 bits[8], and our designs used 5 such FPGAs

Processor	Throughput (MegaPixel)	Hardware Speedup
G4 PPC @450 MHz	4.3	10X
Pentium III @400 MHz	3.3	13X
Pentium III @750 MHz	4.2	10X

Table 2: Throughput of the software implementation of FOA on different processors. Of these processors, only the G4 is easily embeddable.

6 Summary and Conclusions

This paper discussed three evolutionary approaches to mapping FOA circuits to FPGAs. These approaches targetted increased throughput as well as reasonable compile times. All of the circuits we designed were capable of about 46 MegaPixel throughput. This is a 10 times increase over the fastest software implementation on average sized scripts. Moving from highly optimized designs to more general designs, as well as the introduction of runtime reprogrammable circuit elements allowed us to move from device-level configuration to user-level configuration. This in turn allowed us to bypass all of the backend vendor tools, increasing compile times by a factor of 18000, while only sacrificing a 29.6% increase in circuit area. Table 3 shows the changes in circuit size and compiler speed for each of the three approaches. Note that these percentage increases would be much smaller for circuits designed for images with a 1024 pixel width, only 17.5%.

The two level compilation scheme we adopted allows us to generate a generic FOA machine which can be reprogrammed in the manner of seconds. This allowed us to generate a circuit with 10 times greater throughput than the software implemenation, while still allowing the FOA programmer the same flexibility in quickly changing the script being used.

The two-level compilation scheme achieves both high performance and fast compilation times because it *reuses* a *generic* hardware structure that can be customized to implement a specific FOA script. High performance is achieved because the generic FOA image-processing pipline has been carefully designed to meet the needs of basic FOA operations. Fast compilation times are achieved because only a very small amount of hardware (RAM content) is modified, based on the operations that comprise the

Approach	Neighborhood Size (In CLBs)	Size Increase	Compile Time	Compiler Speedup
Direct Synthesis	81	–	10 hours	–
Generic Machine	88	8.6%	30 minutes	$20X$
Generic Machine with runtime config.	105	29.6%	2 seconds	$18000X$

Table 3: Comparison of approaches for FOA mapping. Size increase and compiler speedup are given relative to the first implementation.

FOA script. This is somewhat analogous to the situtation that arises when programming a conventional microprocessor. Here, compilation times are quick (at least relative to general hardware synthesis) because the core hardware (the microprocessor) gets reused in every program generated by the compiler. Rather than synthesizing new hardware operations each time the compiler is run, a 'C compiler (for example) simply implements a given computation by selecting from a set of previously-implemented operations (microprocessor instructions).

Contrast this with the typical configurable-computing application where the entire hardware organization is usually generated from scratch each time the compiler runs, usually involving some form of hardware synthesis. Thus, fast compilation will always be a challenge for configurable computing because it requires a basic tradeoff between design reuse (to reduce compile time) and hardware customization (to achieve high performance). This paper presented a basic two-level compiler strategy that worked because any of the operations found in an FOA script could be implemented with generic computational modules that require only a small amount of customization. Future applications of configurable computing that require fast compilation times will likely need to develop multi-level compilation schemes similar to that described in this paper in order to achieve reasonable compilation times.

7 Future Work

There are many more optimizations that can be done to the circuit implementation to increase throughput and/or decrease circuit size. We are currently working on porting *Supercell* to Xilinx Virtex family FPGAs, and are looking at optimizations that take advantage of new architectural features. We are also looking at modifying the circuit to accept more than one pixel per clock cycle. We have determined that such a modification would add only a small amount of circuitry.

We are currently taking advantage of the generic circuit to rewrite the second-level compiler to make it easily extensible by the end-user. This would make it possible for the end user to add instructions to the compiler in a straightforware, intuitive way. All the user need do is extend a class we provide and give us the details of the neighborhood plane, templates and final calculation. This is an important step forward, as adding instructions to the direct synthesis approach required circuit design. Now, the user need only describe instructions in software, in terms with which they are familiar.

We are also looking into other algorithms to see if the two-level compiler approach we used here would have application to other types of algorithms.

References

[1] S. R. Sternberg, “Automatic image processor.” U.S. Patent 4,167,728, September 1979.

[2] ERIM (Environmental Research Institute of Michigan), Ann Arbor, Michigan, *C4PL Advanced Ptogramming Manual*, 3 ed., February 1993.

[3] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis, and Machine Vision.* PWS Publishing, 1999.

[4] R. M. Haralick and L. G. Shapiro, *Computer and Robot Vision*, vol. Volume I. Addison-Wesley Publishing Company, 1992.

[5] P. Alfke, “Efficient shift registers, lfsr counters, and long pseudo-random sequence generators,” Tech. Rep. XAPP 052, Xilinx, San Jose, CA, July 1996.

[6] P. Bellows and B. L. Hutchings, “JHDL - an HDL for reconfigurable systems,” in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (J. M. Arnold and K. L. Pocek, eds.), (Napa, CA), pp. 175–184, Apr. 1998.

[7] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, “A cad suite for high-performance fpga design,” in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines* (K. L. Pocek and J. M. Arnold, eds.), (Napa, CA), p. n/a, IEEE Computer Society, IEEE, April 1999.

[8] Xilinx, *The Programmable Logic Data Book*, 1999.

Architectures for System-Level Applications of Adaptive Computing

Brian Schott, Chen Chen, Steve Crago,
Joe Czarnaski, Matt French, Ivan Hom[*], Tam Tho, and Terri Valenti

University of Southern California Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203

The mission of the Systems-Level Applications of Adaptive Computing (SLAAC) project is to design and implement a distributed adaptive computing systems architecture. This systems-level focus of SLAAC resulted from the realization that scalability and portability are the two main obstructions preventing innovative Adaptive Computing Systems (ACS) research from being directly useful in deployed real-time environments. Scalability is an issue in that many real-world applications are larger than the modern PCI-based FPGA accelerator. Transitioning from a small proof of concept demonstration to large real-world application is often overlooked in ACS research. Portability has both a hardware and software aspect. Physical form-factor and operating system issues can limit the utility ACS research done in the lab with desktop PCs unless there is a development path to more traditional real-time environments. The SLAAC project seeks to remedy these issues of scalability and portability in ACS systems.

The SLAAC approach to scalability leverages modern cluster-computing techniques to build a scalable parallel ACS system. The SLAAC team has developed a scalable API and runtime library to make it simple to program a cluster of ACS-accelerated workstations. This architecture is called the SLAAC Research Reference Platform (RRP). The Tower of Power (ToP) at Virginia Tech is a good example of an RRP. The ToP has sixteen Pentium II™ PCs each equipped with a WildForce™ board on a sixteen port Myrinet™ switch. A total of 80 XC4062XL FPGAs and memory banks are distributed throughout the platform, and are available as computing resources [1].

The SLAAC approach to portability is to provide source-code compatibility between an RRP cluster and a field-friendly implementation of the identical architecture. The field-friendly implementation is referred to as the Deployable Reference Platform (DRP). Scalable source-code compatibility can be achieved with respect to the host processor application by porting the API and runtime library to a real-time operating system. However, compatibility at the FPGA chip level requires the same ACS board-level architecture for both the RRP and DRP platforms. For this purpose, the SLAAC team has developed two separate ACS accelerator boards that represent the same ACS architecture in the different environments. SLAAC-1 (Figure 1) is a 64-bit PCI board intended for use in RRP workstations. SLAAC-2 (Figure 2) is a mezzanine board designed to be plugged into a modified commercial 6U VME CSPI 2641/S dual PowerPC multicomputer baseboard (Figure 3). There are two SLAAC1-compatible FPGA architectures on the SLAAC2 daughter board, each controlled by one of the two baseboard PowerPCs.

The basic SLAAC architecture is an attached processor system comprised of FPGAs and fast local memories. The concept isn't significantly changed from predecessor reconfigurable computer architectures such as Splash 2 [2] and Wildforce™ [3]. As shown in Figure 4, the SLAAC architecture has four FPGAs — one interface chip (IF), and three user-programmable chips (X0, X1, and X2). The IF chip provides a stable system bus interface for configuring and controlling the user FPGAs from the host processor. IF is booted from an EEPROM that is in-system programmable for software upgrades. The SLAAC architecture is designed to act either synchronously with the host, or asynchronously with DMA channels to and from host memory. DMA logic and SRAM-based FIFOs are implemented within the IF chip. IF has a memory bus to allow the host to bypass the user FPGAs and access the user memories directly.

The three user-programmable FPGAs are organized in a ring structure. X0, X1, and X2

[*] University of Southern California Information Sciences Institute,
4676 Admiralty Way, Suite 1000, Marina del Rey, CA 90292

also share a common “crossbar” bus. In order to track modern processor architectures, the data paths are 72-bits wide, allowing for 64-bits of data and an 8-bit tag on each path. Six additional handshake lines that provide dual pair-wise connections among X0, X1, and X2 are not shown. The X0 device is a Xilinx XC4085XLA FPGA, and both X1 and X2 are XC40150XV FPGAs for a total of 750K user logic gates. In addition, the SLAAC architecture has ten 256Kx18 synchronous SRAMs that feature zero-bus turnaround. A read or write is permitted every cycle up to 100MHz. SLAAC is designed to approximate ten “Splash-2-like” single-memory systolic processing elements. Pipelining is improved in X1 and X2 by connecting the memories to the pads along the top of the FPGA, the crossbar along the bottom, and the left and right ring connections along the left and right pads of the chip respectively. Both X1 and X2 footprints are identical so that the same configuration can be easily replicated without redundant synthesis. Unlike the Splash 2 and WildForce™, X0 in the SLAAC architecture is designed to control both ends of the systolic array. X0 acts as the data manager and pre/post processor for systolic data streams running through the system.

The SLAAC team has completed development of the SLAAC-1 and SLAAC-2 boards and is performing several defense application demonstrations on these platforms this year. Application areas include radar target recognition, infrared target queuing, electronic counter-measures, sonar signal processing, and multi-dimensional image processing. Future work includes a migration to Virtex-based SLAAC architectures and additional application demonstrations.

Reference

[1] P. Athanas, M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, “*Implementing an API for Distributed Adaptive Computing Systems*,” submitted to the Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.

[2] D. A. Buell, J. M. Arnold and W. J. Kleinfelder, “*Splash 2 FPGAs in a Custom Computing Machine,*” IEEE CS Press, Los Alamitos, CA, 1996.

[3] Annapolis Microsystems Incorporated, “*WildForce User’s Guide,” 1998.*

[4] W-K. Su, R. Sivilotti, Y. Cho, and D. Cohen, “*Scalable, Network-Connected Reconfigurable, Hardware Accelerators for an Automatic-Target-Recognition Application*,” Web page, http://www.myri.com/research/darpa/atr-report.pdf. May 1998.

This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government.

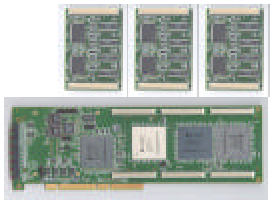


Figure 1.
SLAAC-1

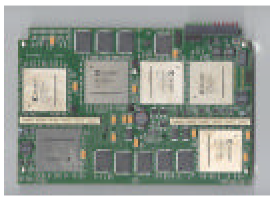


Figure 2.
SLAAC-2

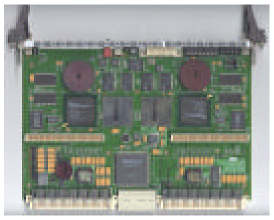


Figure 3.
CSPI 2641/S

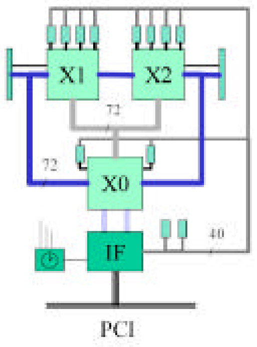


Figure 4.
SLAAC Architecture

Co-design of Software and Hardware to Implement Remote Sensing Algorithms

James Theiler, Jan Frigo, Maya Gokhale, and John J. Szymanski

Space and Remote Sensing Sciences, Los Alamos National Laboratory, Los Alamos, NM 87545

ABSTRACT

Both for offline searches through large data archives and for onboard computation at the sensor head, there is a growing need for ever-more rapid processing of remote sensing data. For many algorithms of use in remote sensing, the bulk of the processing takes place in an "inner loop" with a large number of simple operations. For these algorithms, dramatic speedups can often be obtained with specialized hardware. The difficulty and expense of digital design continues to limit applicability of this approach, but the development of new design tools is making this approach more feasible, and some notable successes have been reported. On the other hand, it is often the case that processing can also be accelerated by adopting a more sophisticated algorithm design. Unfortunately, a more sophisticated algorithm is much harder to implement in hardware, so these approaches are often at odds with each other. With careful planning, however, it is sometimes possible to combine software and hardware design in such a way that each complements the other, and the final implementation achieves speedup that would not have been possible with a hardware-only or a software-only solution. We will in particular discuss the co-design of software and hardware to achieve substantial speedup of algorithms for multispectral image segmentation and for endmember identification.

Keywords: co-design, configurable computing, field-programmable gate array (FPGA), k-means, endmembers

1. INTRODUCTION

Studying algorithms requires thinking in several ways:
creatively, to discover an idea that will solve a problem;
logically, to analyze its correctness;
mathematically, to analyze its performance; and
painstakingly, to express the idea as a detailed
sequence of steps so it can become software.

– Christopher Van Wyk[1]

An algorithm is a procedure for getting something done. The specification of an algorithm must be detailed and precise; it is an exact and painstaking science. But the design of algorithms is an art. The recent emergence of reconfigurable hardware, and in particular the commercial availability of Field Programmable Gate Array (FPGA) chips and boards,[2–4] has added a new dimension to the design space. By exploiting the inherent fine-grained parallelism available in programmable hardware, one can obtain speedups of one to two orders of magnitude over what is achievable on a general purpose serial processor – and this is in spite of the fact that the FPGA clock speed is typically an order of magnitude slower than the microprocessor.[5] Unfortunately, the price paid for this higher performance is a much longer design cycle. There is currently a vigorous research and development effort to create design tools for reducing the length of this cycle,[6–8] but the practical upshot for scientific research (where one, or at most a few, of the implementations will be deployed) is that it is not cost-effective to design a full-up hardware implementation for any but the simplest algorithms.

On the other hand, even simple algorithms have their place. The lowly dot product is an example that is as widely applicable as it is mathematically trivial. An efficient dot product can be implemented on an FPGA,[9] but the dot product is also a good example of an "algorithm" which is usually just one part of the real algorithm (eg, an adaptive matched filter[10,11]) that you may want to implement for your remote sensing data analysis.

E-mail: {jt,jfrigo,maya,szymanski}@lanl.gov

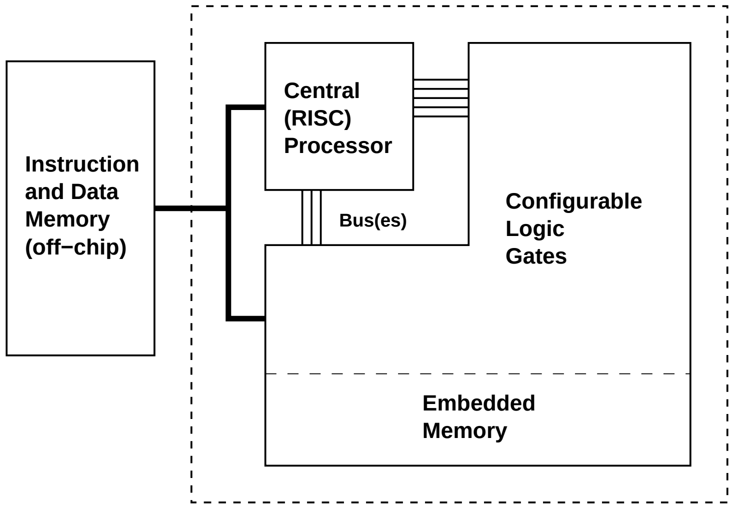


Figure 1. Typical Configurable System on a Chip (CSoC) architecture. A central RISC processor is built into the same chip as the logic gates and embedded memory modules, and both have access to local on-board, but off-chip, memory. The size and number of buses connecting the central processor to the configurable logic can itself be configured to achieve a bandwidth that is appropriate for the application.

Hardware implementation is sometimes seen as a brute force approach to speeding up algorithms. Generally, to implement an algorithm in hardware requires a lot more effort than to implement it in software. And generally, to fit an algorithm into hardware, certain approximations, truncations, and/or simplifications must be made. These approximations must be traded against the raw speed provided by the fine-grained parallelism in a hardware implementation.

One very nice example of a co-design was developed by Porter *et al.*[12,13] It employs a genetic algorithm in software that specifies the settings of hardware registers to direct a generic image processing operator that is implemented in an FPGA. Compared to the original all-software version of this algorithm,[14] this hardware-accelerated system has a much smaller operator set, but it makes up for this with a search through operator space that is roughly two orders of magnitude faster. This example also illustrates the need to decompose an algorithm so that there is a simpler part (often an "inner loop") that can be sped up in hardware in such a way that doesn't interfere with potentially complicated branching logic (in the "outer loop"). This complicated outer loop logic is what allows you to design sophistication into your algorithm, sophistication that may enable further speedup, or that may simply be necessary to achieve a sophisticated goal.

For algorithms that employ substantial software and hardware components, the biggest challenge is the communication bandwidth between them. One answer to this bottleneck is a hybrid processor that combines a traditional microprocessor and programmable logic on the same chip.[15–17] (see Fig. 1.) Such hardware is just now becoming commercially available.[18–27] We have recently used the Altera Excalibur board with a soft-core 32-bit NIOS RISC processor[18] as part of a co-design for k-means clustering.[28] Unfortunately, the promise of seamless integration of a fast central processor surrounded by acres of programmable logic remains to be fulfilled. We achieved a 15% speedup, but were limited by slow memory access and a slow central processor – since the processor on the Excalibur is implemented as a "soft core" it is constrained to run at the same clock speed as the programmable logic itself. With a faster hybrid processor (eg, 10x faster than configurable circuits) or dual port memories that both processor and configurable circuits can access, then we would expect 10x speedup over sequential algorithms. We furthermore extrapolated the expected performance to the Xilinx-PowerPC hybrid[19] and predicted the possibility for two orders of magnitude acceleration.

In the following two sections, we will discuss the possibilities for further co-design opportunities – these simultaneously employ "smart" software and "brute force" hardware to speedup two problems of interest in remote sensing: image segmentation and endmember estimation.

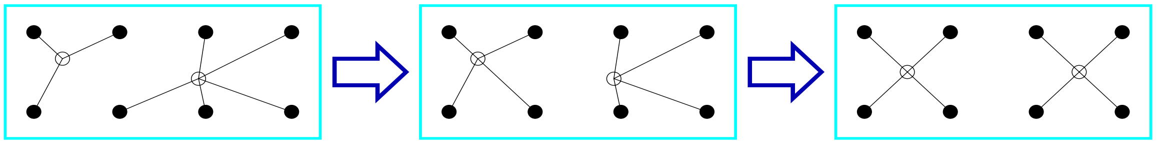


Figure 2. The idea of the k-means algorithm. Data points are represented as filled circles, and centers are open circles. The first step assigns each data point to the center that it is closest to. The second step (re)locates the centers to be the mean value of all the points in the cluster.

2. CLUSTERING AND SEGMENTATION

Although the first published references to the k-means algorithm can be traced to the mid-1960's,[29–32] the basic idea for the algorithm is quite simple (see Fig. 2) and is sometimes called the "generalized Lloyd algorithm" since it is a vector version of Lloyd's original scalar quantization algorithm which was developed in the late-1950's.[33] The fine-grained parallelizability of the algorithm has long been noted,[34] and this makes it a particularly attractive target for hardware implementation. The k-means algorithm is a kind of special case of the Expectation Maximum (EM) algorithm introduced and analyzed by Dempster *et al.*[35] It can be proved to result in better approximations to a local optimum with every iteration; a global optimum, unfortunately, is not guaranteed. The literature on clustering algorithms is quite vast, and there are many reviews[36–41]; partly this is because there is such a wide range of applications.

While multi- and hyper-spectral imagery provide unprecedented amounts of information about a scene, it also presents the image analyst with something of a headache. It is not humanly possible to look at all those channels at the same time. By clustering the data, one provides not only a substantial (albeit lossy) compression, but also a "picture" in which pixels with similar spectral signature are identified by a unique (and usually false) color.

As well as reducing the data for quicklook views, clustering also provides an organization of the data that can be useful for other processing downstream. For example, Kelly and White[42] employ clustering as a preprocessing step to speed up interactive/exploratory manipulations of large data sets. Also, Schowengerdt[43] suggests the use of image segmentation for change detection: a change in the segmentation is more likely to indicate an actual change on the ground, since the segmentation is relatively robust to changes in sensor performance and atmospheric conditions. Several authors have shown that clustering the data beforehand improves the performance of algorithms which attempt to "learn" features from a small number of labelled examples.[44–49] The use of clustering for image restoration has also been explored in some detail; see Sheppard *et al.*[50] for a recent review. For remote detection and characterization of gaseous plumes, the ground scene ceases to be the signal of interest, and becomes instead the clutter. Clustering provides a way to reduce this background clutter, because the within-class variance of a segmented image can be made much smaller than the overall variance of the image as a whole.[51] The issues of clustering and pixel mixing are somewhat at odds with each other, but a paper by Stocker and Schaum[52] points to one approach for combining them.

In the subsections that follow, we will describe both previous and ongoing work to modify the traditional k-means algorithm so as to speed it up with the aid of programmable logic hardware.

2.1. Low-level variants of k-means

One of the first, and most straightforward, ways to alter a floating-point serial-processor algorithm is to truncate the precision of the data and/or of the intermediate computations, and to do the computation in fixed-point arithmetic. We found in Leeser *et al.*[53] that good clusterings could be achieved, even with considerable truncation of the data. We found that it was advantageous to allocate two more bits of precision to the centers than to the data, but this is relatively cheap since there are few centers and many data points.

Truncating the precision of the data still keeps the algorithm more or less intact; a less obvious way to alter the algorithm is to incorporate approximations which are more amenable to hardware. At the core (and in the inner loop) of the k-means algorithm is the computation of distance between the centers and the data points. By altering the distance metric itself, we were able to achieve a hardware implemention which greatly enhanced speedup.

In one dimension, distance between two points is straightforwardly given by the absolute difference between their coordinate values. In higher dimension, the distance between two points can be expressed in terms of the

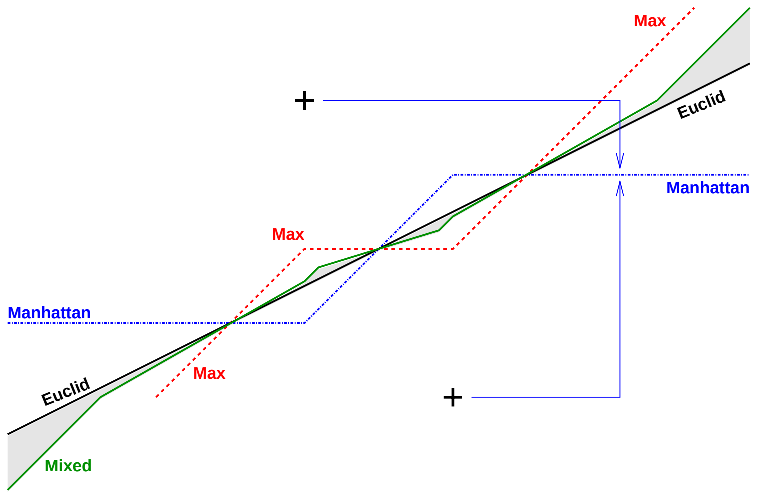


Figure 3. The boundary between classes with centers indicated by crosses in the above figure depends on the distance metric. For instance, the two arrows leading from the class centers to the Manhattan boundary are both the same length (in the Manhattan sense). The jagged line that roughly follows the "correct" Euclidean boundary is produced by the mixed distance measure with $\alpha = 0.25$. Points which fall inside the shaded area between the mixed and the Euclidean boundary would be misclassified by the mixed distance metric. Note how much more accurately the mixed boundary (compared to the Max or Manhattan boundaries) approximates the true Euclidean boundary.

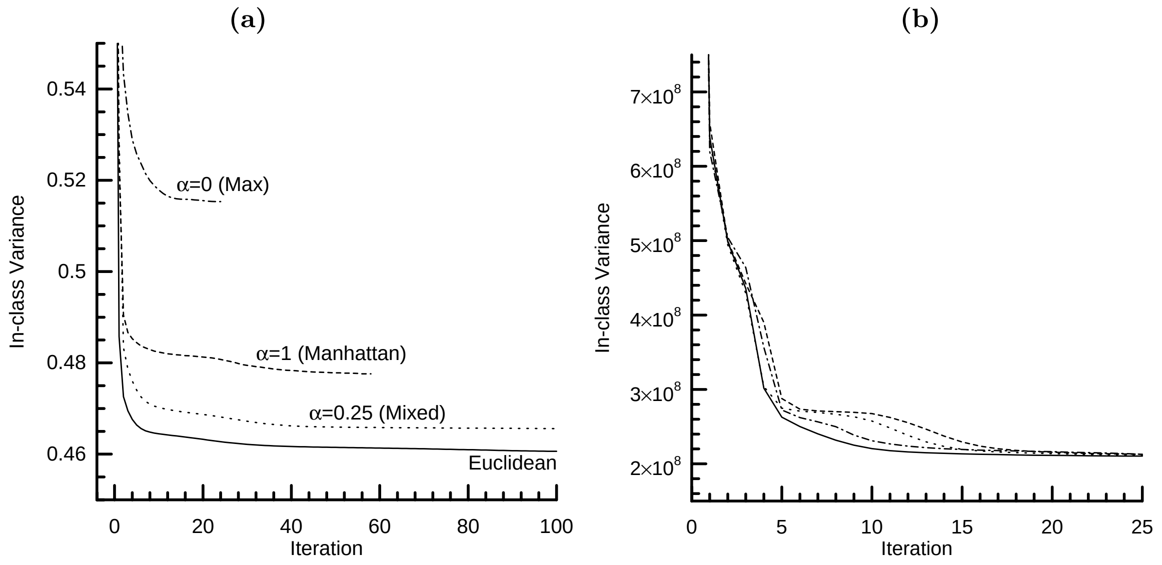


Figure 4. Comparison of k-means performance for different distance metrics. **(a)** This is for $N = 10^5$ points randomly distributed in a $D = 8$ dimensional unit cube, and clustered into $K = 8$ classes. It is evident that the in-class variance is minimized if the "correct" (but expensive) Euclidean distance is used. The Max and Manhattan metric are two alternatives which are much cheaper to implement in hardware, but which do not perform as well as the Euclidean. The Mixed metric is a linear combination of the Max and Manhattan metric; this is still cheaper than the Euclidean metric, but performs nearly as well. **(b)** A multispectral image was created from an AVIRIS hyperspectral datacube[54] (flight number f960323t01p02_r04_sc01) resampled to produce $D = 10$ channels corresponding to bands available on the MTI satellite.[55] This $N = 512 \times 614$ image was clustered into $K = 8$ classes. The theoretical differences between the different distance metrics, seen in panel (a), are much less evident in real data.

one-dimensional coordinate-wise distances. A general family of such distances is parameterized by p:

$$\|\mathbf{x}-\mathbf{c}\|^p = \sum_i |x_i - c_i|^p. \tag{1}$$

In this family, $p = 2$ corresponds to Euclidean distance, which is a rotationally-invariant measure which furthermore corresponds directly to the in-clsss variance which the k-means algorithm attempts to minimize. Although $p = 2$ is theoretically optimal, two other members of this family, $p = 1$ and $p \to \infty$, are particularly cheap to compute. The Manhattan distance ($p = 1$) is the sum of absolute values of coordinate differences, while the Max distance ($p \to \infty$) is simpy the maximum of all the absolute values of the coordinate differences. Since the error made by the Max distance is usually in the opposite direction as the error made by the Manhattan distance (see Fig. 3), we introduced a better approximation[56] which can be achieved with a simple linear combination of these two:

$$\|\mathbf{x}-\mathbf{c}\|_\alpha = \alpha\, d_{\text{Manhattan}}(\mathbf{x},\mathbf{c}) \;+\; (1-\alpha)\, d_{\text{Max}}(\mathbf{x},\mathbf{c}) \tag{2}$$

We found that $\alpha = 0.25$ was near the optimal value over a wide range of dimensions and furthermore, being a (negative) power of two, was advantageous for hardware implementation. The relative performance of the different distance metrics is shown in Fig. 4, and discussed in more detail elsewhere.[56–58]

2.2. Block updates of centers

In the classic k-means algorithm one sweeps through all the data, assigning each data point to the center it is closest to. At the end of this sweep, the centers are recomputed by taking the mean value of all the points with the label corresponding to that center. However, faster convergence is generally observed when the centers are updated on the fly; that is, centers are recomputed after each new (re)assignment. This is sometimes called the "continuous k-means" algorithm,[60] but the basic idea has been around for a long time.[39] Fig. 5 compares the convergence rate for the standard ($B = N$) and on-the-fly ($B = 1$) versions of the k-means algorithm for some example data.

Unfortunately, updating the centers on the fly is hard to do in hardware; it requires (among other things) a division operation for each data point, instead of just one for each center at the end of the sweep through all the data points. In designs by Leeser *et al.*[56–59] the division operation was offloaded to the general-purpose processor. But it would not be feasible to do this every time a data point was reassigned.

The on-the-fly approach also de-parallelizes the algorithm. In the standard approach, the assignment of a point to a class depends only on fixed centers (so all points could be assigned independently – and in parallel if desired). But the update of centers every time a data point is reassigned will affect subsequent class assignments. A branch condition has been introduced into the "tight inner loop" and the hardware approach is not able to take advantage of this software speedup.

To achieve the software speedup provided by dynamically updated centers, while still using hardware for the main inner-loop acceleration, we can employ a compromise in which class assignments are performed on blocks of B pixels at a time. The centers are only updated between blocks. We see in the experiments in Fig. 5 (on both synthetic and real data) that as long as $B \ll N$, the improvement obtained by on-the-fly updates will still be obtained for relatively large values of B. Fig. 6 shows the dataflow for the implementation reported by Gokhale *et al.*[28] for k-means with blocks of B assignments at a time.

Using this implementation, we performed an experiment to determine the effect of block size B on the overall speed of the k-means co-design. We know that smaller block size leads to faster convergence in software (see Fig. 5), but we expect that larger block sizes will be more efficient in hardware, so we were interested in finding an optimum.

We used random data with $D = 8$ spectral bands, and $b = 8$ bits per point, and we clustered into $K = 8$ classes. We evaluated the algorithm by increasing the size of B, but we found that for $B > 8$, the execution speedup was essentially independent of B, and essentially the same value that we reported earlier,[28] about fifteen percent – not enough yet to be practical.

The bottleneck in this computation is the bandwidth between the central processor and the configurable logic. Using the parallel I/O or memory mapped standard interfaces on the Excalibur Board, we found that it takes 11 cycles to send one 32-bit word.[28] This is a fundamental limitation of the RISC processor (reduced instruction set means more instruction cycles!) being soft-coded into the configurable logic, and therefore running at the same rate

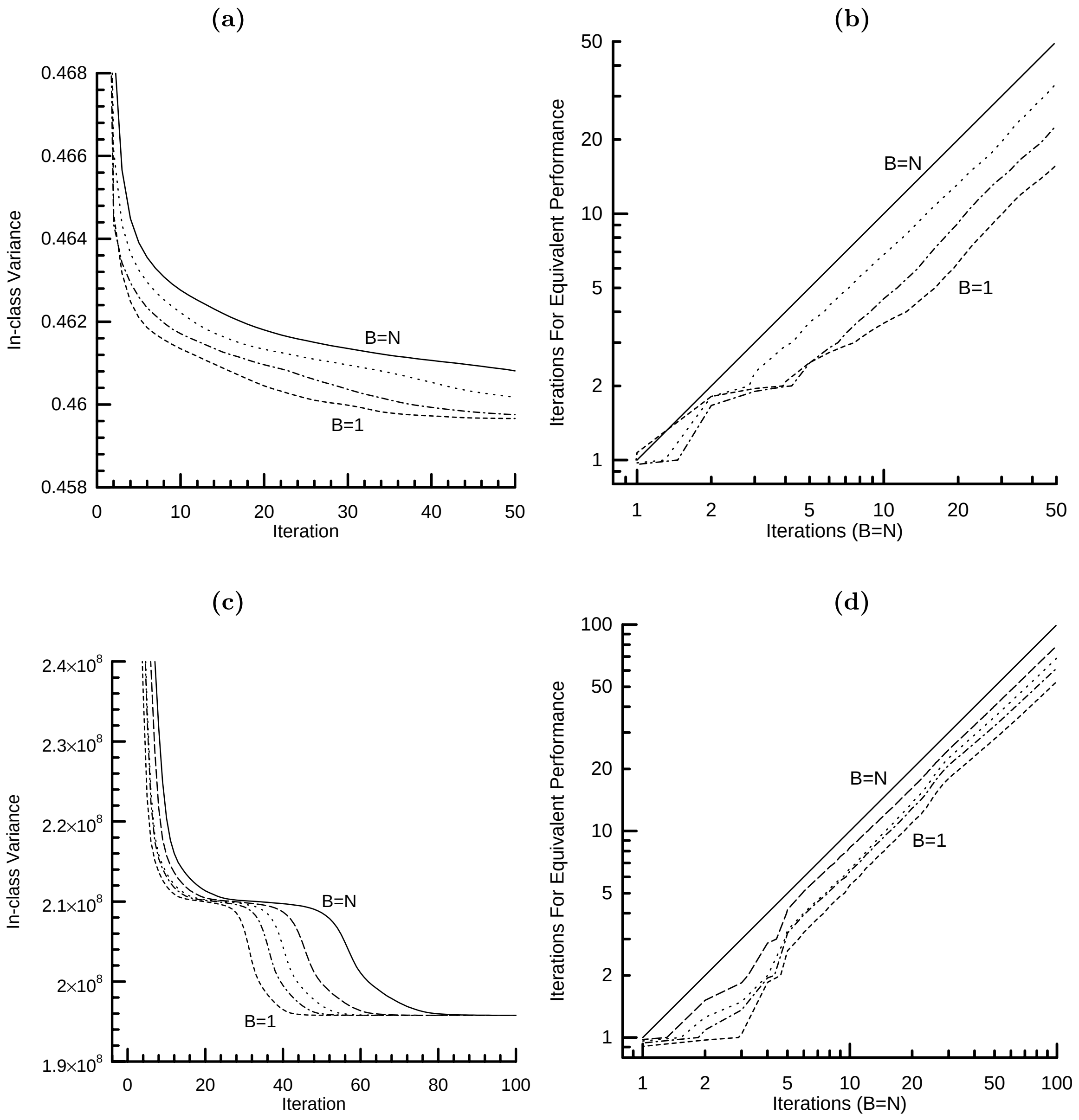


Figure 5. Performance of k-means clustering when updating centers after blocks of B data points. $B = N$ (solid line) corresponds to the traditional k-means algorithm, with centers updated after a full sweep through the data; $B = 1$ (dashed line) corresponds to updating centers "on the fly" (ie, every time a point is determined to change classes). **(a)** For uniformly distributed random data (see caption of Fig. 4(a)), updating on the fly is seen to produce faster convergence. The performance for intermediate values of B are shown with the dotted ($B = 5 \times 10^4$) and dashed-dotted ($B = 10^5$) lines. **(b)** Here, this faster convergence is explicitly shown – the vertical axis indicates the number of iterations required, for different values of B, to achieve the same performance as was achieved for the number of iterations shown on the horizontal axis using the standard $B = N$ approach. In this case, the on-the-fly approach produced the same quality clustering with roughly a quarter of the iterations required by the standard approach. **(c,d)** Same as in panels (a,b), but for the multispectral image data described in the caption to Fig. 4(b). Again, the on-the-fly approach was more efficient, but in this case, the gain was smaller than for the uniform random data. The intermediate values of B shown here are $B = 5 \times 10^4, 10^5, 2 \times 10^5$; for comparison $N \approx 3 \times 10^5$.

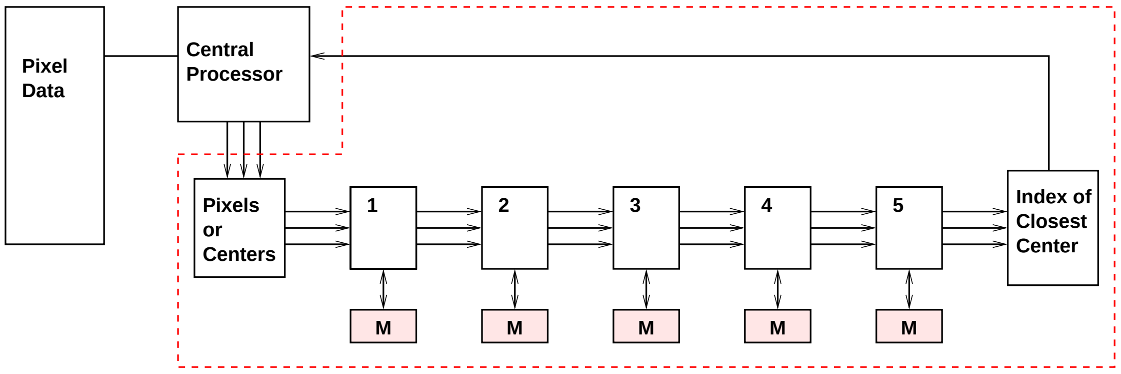


Figure 6. Data flow for k-means implementation on the hybrid processor. The central processor sends out a control bit to the hardware in order to indicate that the following data is center data or pixel data. When center data is indicated, the centers are loaded via the configurable logic to the embedded memory. When pixel data is selected the processor then sends out the block of B pixels. The index of the class which represents the minimum distance is returned to the central processor and it computes the new class centers, which it reloads into the embedded memory. Then it sends another block of B pixels, *etc.* until all the pixels have been evaluated.

as the rest of the chip. If the RISC processor were running on a faster clock (which is the ultimate goal for these hybrid systems), then the 11 cycles would go a lot faster and would not produce the problem they currently do.

A better way of sharing data between the processor and configurable logic is required. We have recently become aware of one such method, peculiar to this chip, that yields a latency of only 3 cycles to fetch data from memory for the configurable logic. This method observes the data and address bus as the software process is reading data and fetches data per a prescribed address map when a particular address location is observed. This method applied to the Kmeans algorithm could obtain an appreciable speed up of 3 to 4 times over our first mapping attempt.

2.3. Parameterizing the hardware implementation

The importance of parameterizing the hardware implementation is provided by an example given by Leeser *et al.*[61,59] There, an earlier implementation of k-means had hardcoded $K = 8$ clusters, $D = 10$ channels, and $b = 12$ bits per channel. A parameterized version was developed which treated these variables as parameters that could easily be changed. This obviously provides more flexibility for the user, but it is interesting to note that this flexibility did not come at the cost of decreased performance. For the same choice of K, D, and b, the parameterized design was actually smaller and virtually the same speed as the hardcoded design; furthermore, the place-and-route time was a factor of three smaller for the parameterized model.

2.4. Further approaches

One variant of the updating by blocks approach was suggested by Domingos and Hulten[62]; here a different block size is used at each iteration, with the size of the block based on estimates both of the misclassification error and of the centroid location error. An attempt is made to use large enough blocks to mimic (to within a prescribed ϵ) the behavior of the $B \to \infty$ convergence, but with $B \ll N$ points.

Kanugu *et al.*[63] describe a very sophisticated application of k-d trees to speed up the implementation of the k-means algorithm. The work achieves near order of magnitude speedup, at least for low-dimensional data, without introducing *any* approximations to the standard k-means algorithm. This is a "feature" but unfortunately it also makes it difficult to combine this approach with other speedups that do alter the basic algorithm. Translating the approach to a hardware implementation would be a challenge, to say the least.

The standard application of k-means to multispectral imagery treats the individual pixels as independent. A contiguity-enhanced approach has been described[64] which clusters the data in a way that produces segmentation with only slightly larger in-class variance, but substantially larger spatial contiguity for the classes. Ideally, we would like to impose this contiguity bias from the software, without altering the hardware design. This would give us the

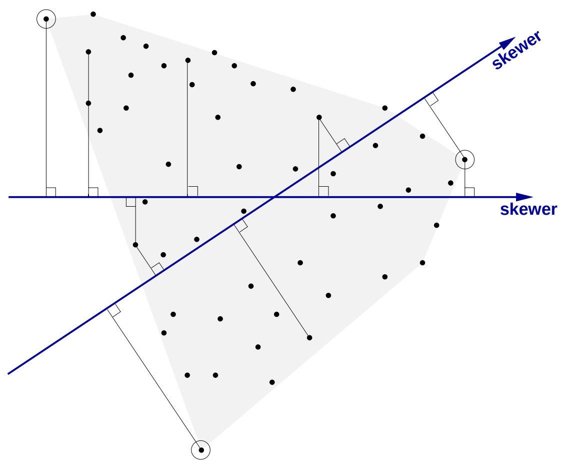


Figure 7. The PPI algorithm works by projecting points in the data set onto random skewers. For each skewer, two extreme points are identified, and their pixel purity index is incremented. In the figure above, the circled points are identified as candidate endmembers in the full space because their projection onto one or both of the skewers is extremal. The gray region indicates the convex hull of the points; only points on the convex hull are identified by the PPI algorithm.

most flexibility, but the contig-k-means algorithm is currently designed so that the bias is built into the distance function (which is computed in hardware). So adapting this problem to a co-design would also be a challenge.

When the desired number of clusters K is large, the standard k-means algorithm has a very slow converge rate, and many more iterations are required than would be for smaller values of K. One common approach is provided by the classic Linde-Buzo-Gray[65] approach; one begins with a small number of classes, and on subsequent iterations increases K. A systematic splitting algorithm was advocated by Fränti *et al.*[66] for faster clustering. Implemented in software, these algorithms achieve speedup in two ways: one is that fewer distance computations are required per data point at least for the early iterations, and the other is that fewer iterations are required. To take advantage of the first effect in hardware, one would have to re-design the configurable logic. But the fact that fewer iterations are required suggests that even a design that computes the sometimes-than-necessary K distance computations on every iteration will achieve a performance gain.

Once a clustering has been found, a split-and-merge algorithm[67] can be used to further improve the in-class variance by overcoming the problem of local minima. This will require some sophistication in the software to decide which clusters to split and which to merge, but with the distance computations farmed off to the highly parallel configurable logic, a good co-design could achieve the speedup benefits of both the hardware and the software, while at the same time achieving gains in the quality of the clustering.

3. ENDMEMBERS AND PIXEL PURITY INDEX

A number of algorithms have been proposed over the last decade for finding so-called endmembers in multispectral data.[68–78] The rationale behind endmembers is that a scene contains relatively few distinct materials and that much of the pixel-to-pixel variation in a multispectral image cube can be explained by the assumption that the pixels are mixtures of these distinct materials. The endmembers are the spectral signatures of these distinct materials, and the spectra of the rest of the pixels can be expressed as linear combinations (with positive weights, summing to unity) of these endmembers.

Endmembers are like principal components in that they provide a basis set for describing the rest of the data. But, because the endmembers are taken from the data themselves, and because the linear combinations are restricted to non-negative coefficients which sum to one, these endmembers are expected to provide a more "physical" basis than principal components.

The Pixel Purity Index (PPI) is one algorithm for identifying emdmembers from a multispectral dataset. The approach is to generate a large number n of random D-dimensional vectors (called "skewers" – see Fig. 7) and to

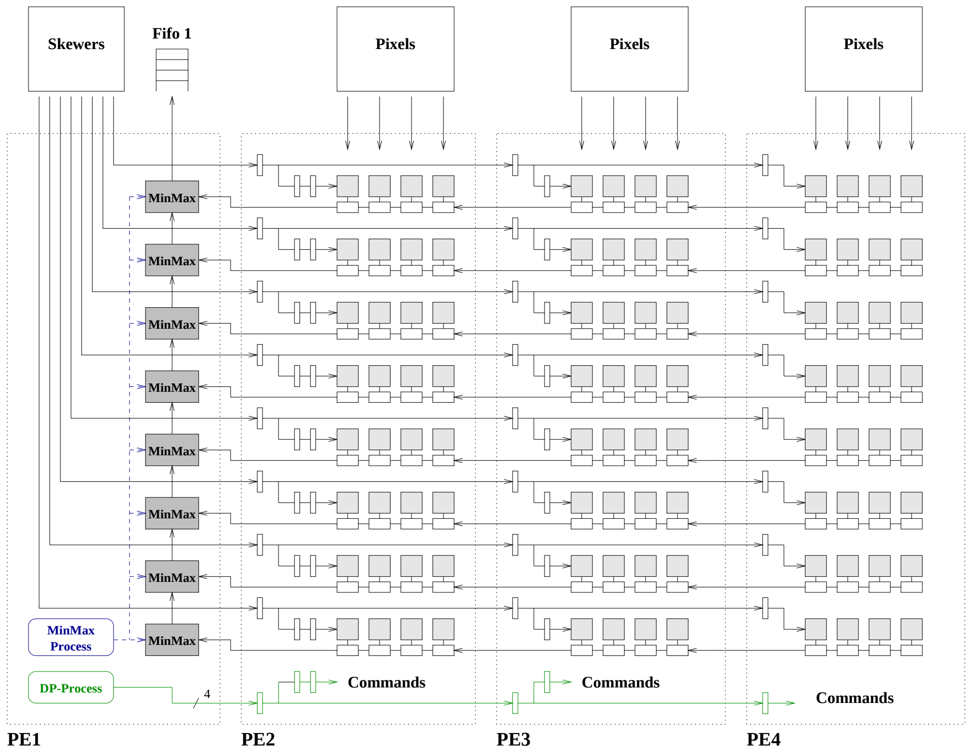


Figure 8. Architecture of an implementation of PPI on the Wildforce card.[79] The Wildforce contains four processing elements (PE1, PE2, PE3, PE4), each a distinct FPGA chip. This design uses three of the elements for computing dot products and the fourth for bookkeeping: routing skewer coefficients to the dot product components, computing extrema of the dot products and keeping track of the indices of pixels that produces those extreme, *etc.*

project the D-dimensional data onto them. A dot product is computed for every data point against every skewer, and the data points which correspond to extrema (or near extrema) in the direction of the skewer are identified, and placed on a list. As more skewers are generated, this list grows; the number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the most pure, and a pixel's count provides its "pixel purity index".

Lavenier *et al.*[79] describe an implementation of the PPI algorithm on a reconfigurable chip. See Fig. 8. A software approach for fast PPI was discussed by Theiler *et al.*[80] which expanded a small set of "direct dot products" into a larger set of "derived dot products." Although this approach was originally designed as a software speedup, and as such was a "competitor" to the hardware implementation, Fig. 9 shows how the software approach could be incorporated into the hardware design, combining the speedups from both approaches.

The pixel purity index was originally conceived[68] not as the full solution to the endmember problem, but as a a guide to be employed in an interactive way by a competent remote sensing scientist in conjunction with a good spectral library. A fully automated endmember identification algorithm might be desirable, but (with this approach, at least) is not something that one would want to design directly into hardware. However, the full endmember problem provides many opportunities for a co-design in which the software is responsible for directing the actual PPI in hardware, and for interpreting its results. The PPI algorithm, for instance, does not by itself identify a final list of endmembers; simply taking the $D + 1$ "most pure" pixels can lead to degenerate sets in which some of the endmembers are nearly identical. The choice of which pure pixels to choose as final endmembers is a postprocessing step that is better designed in software, where adjustments and modifications are easier to make. For instance, an otherwise expensive methodology with possibly complicated statistical properties (such as the archetypes of Cutler and Brieman[70]) could be more cheaply computed using the pixels with high purity index for initialization. The use of PPI to initialize the hypervolume maximizing algorithm (NFINDR) of Winter[78] has been described previously.[80] A closer interaction of the software and hardware would permit more complicated strategies for choosing the skewer directions based on endmembers that have been found so far – this might entail both exploration (in directions

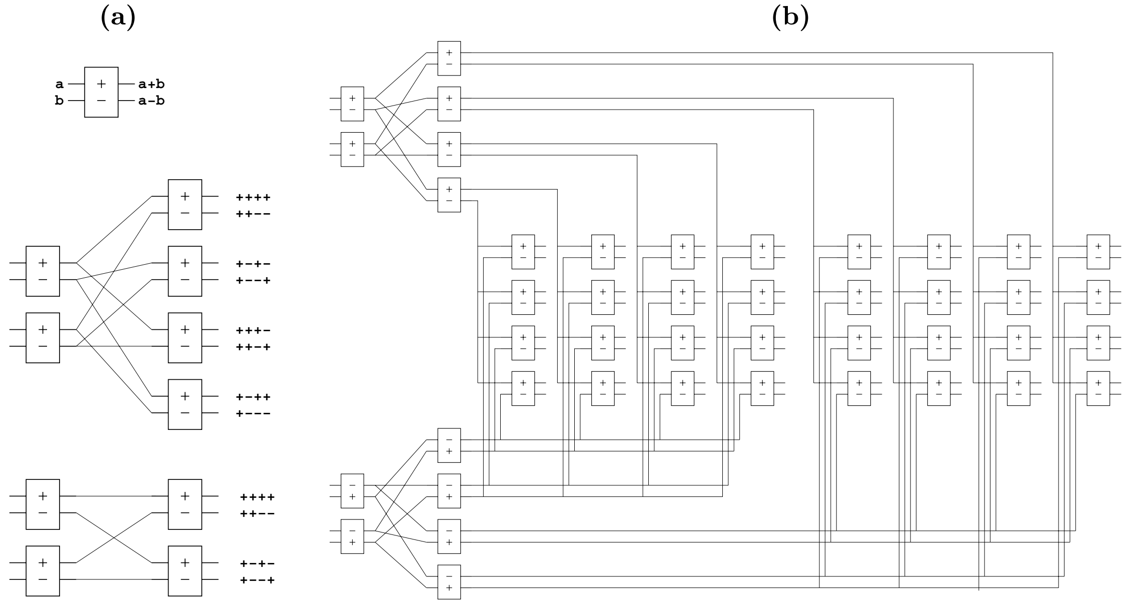


Figure 9. A schematic circuit for extending the PPI algorithm implemented in Fig. 8 to achieve eight-fold gain in the number of skewers without computing any more direct dot products. **(a)** The main building block is a module that takes two inputs and produces two outputs, corresponding to the sum and to the difference of the inputs. Combined into a circuit, six of these modules can take four inputs and produce eight outputs corresponding to all the sums and differences of the inputs. The positive and negative values of these eight sums correspond to the sixteen corners of a four-dimensional hypercube. Using only four modules, one can produce four outputs corresponding to the eight "alternate" corners of the four-dimensional hypercube. These alternates are all a Hamming distance of at least 2 from each other. **(b)** With 44 modules, one can leverage 8 inputs (direct dot products) into 64 outputs (derived dot products) corresponding to the alternate corners of an eight-dimensional hypercube. With this circuit element, one can transform the result of 8 direct dot products into 64 derived dot products. Applied to the Pixel Purity Index algorithm, this permits eight times as many skewers with only a fixed cost in terms of chip area. With respect to the design in Fig. 8, this circuit would fit between the dot products on PE 2-4 and the min-max computations on PE1. The eight min-max units on PE1 would have to be expanded to sixty four min-max units, but no extra dot products need be computed.

roughly perpendicular to existing endmembers) and refinement (with many nearly parallel skewers in the direction of existing endmembers).

4. CONCLUSION

If an algorithm is a procedure for getting something done, then much of the art in algorithm design depends on knowing about that "something." In contrast to the detailed specifications that go into the *implementation* of an algorithm, the ultimate *aim* of an algorithm is often just a little bit fuzzy. Some classical algorithms (eg, sorting an array, computing a Fourier Transform, or finding the convex hull) are indeed well-specified, but the algorithms that one needs in the real world, particularly in the real world of remote sensing, do not always have such mathematically well-defined goals.

While mathematical expressions of those goals are an important starting point, one must sometimes step back and remember what the real goals are. Clustering puts spectrally similar pixels into the same class, and has the effect of reducing the within-class variance. But the mathematical goal of finding *the* partition with the global minimum of within-class variance is a combinatorially difficult problem. On the other hand, by using an iterative algorithm (k-means) with a simplified distance measure (Manhattan) on precision-truncated data, one can very rapidly compute an approximation to the optimal solution. This approximation may be suboptimal from the mathematical point of view, but it does achieve the real goal – that spectrally similar pixels be classified together. More important than squeezing the last percentage point of optimality from the mathematical constraints is to reconsider the physical phenomena (such as correlations between spectral channels or the tendency for contiguous pixels to be in the same class) that drive the mathematical formulation.

By the same token, while the concepts of convex geometry provide useful insight into the problem of detecting endmembers in imagery, we have to keep in mind that the endmember concept itself is an approximation: real scenes are not really composed of a small number of distinct materials, nor are pixels exact linear combinations of of these materials. The convex hull is a useful geometrical visualization of what we desire to estimate from our data, but we do not literally want to find the convex hull of a million points in a hundred dimensional space.

Remote sensing is a complicated science; and algorithms for remote sensing retrievals will necessarily be complicated as well, if they are to achieve a useful level of physical fidelity. Complicated algorithms do not fit well in hardware, and are a waste of valuable real estate that could be used for further fine-grained parallelism. We do believe that reconfigurable hardware has a place in remote sensing computation, particularly for real-time processing. But to produce fast implementations of complicated and flexible algorithms will require the simultaneous design of some components that run well in massively parallel mode on reconfigurable hardware, some components that run well on general purpose serial processors, and appropriate interconnections that keep both talking to each other without overloading the limited bandwidth between them.

ACKNOWLEDGMENTS

We are pleased to acknowledge many stimulating conversations with Miriam Leeser and with Dominique Lavenier on this topic. This work was supported by the United States Department of Energy, through the Deployable Adaptive Processing Systems (DAPS) project at Los Alamos National Laboratory.

REFERENCES

1. R. Sedgewick and C. J. Van Wyk, *Algorithms in C++*, Addison-Wesley, Reading, Massachusetts, 1998.
2. Xilinx, Inc. `http://www.xilinx.com`.
3. Altera Corporation. `http://www.altera.com`.
4. Annapolis Micro Systems, Inc. `http://www.annapmicro.com`.
5. J. Villasenor and W. H. Mangione-Smith, "Configurable computing," *Scientific American* , pp. 66–71, June 1997.
6. P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, and D. Zaretsky, "A MATLAB compiler for distributed, hetergeneous, reconfigurable computing systems," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, eds., pp. 39–48, IEEE Computer Society Press, 2000.

7. M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA computing in the Streams-C high level language," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, eds., pp. 49–58, IEEE Computer Society Press, 2000.
8. B. Draper, W. Najjar, W. Bohm, J. Hammes, B. Rinker, C. Ross, M. Chawathe, and J. Bins, "Compiling and optimizing image processing algorithms for FPGAs," in *International Workshop on Computer Architectures for Machine Perception (CAMP)*, 2000.
9. M. P. Caffrey, A. Begtrup, J. Layne, T. Nelson, S. Robinson, A. Salazar, J. J. Szymanski, and J. Theiler, "High performance signal and image processing for remote sensing using reconfigurable computers," *Proc. SPIE* **3807**, pp. 142–149, 1999.
10. P. V. Villeneuve, H. A. Fry, J. Theiler, and B. W. Smith, "Improved matched filter detection techniques," *Proc. SPIE* **3753**, pp. 278–285, 1999.
11. A. Nelson and K. McCabe, "Flexible architecture for hyperspectral image processing on reconfigurable computers," Tech. Rep. LA-UR-01-2900, Los Alamos National Laboratory, 2001.
12. R. Porter, M. Gokhale, N. Harvey, S. Perkins, and C. Young, "Evolving network architectures with custom computers for multi-spectral feature identification," in *3rd NASA/DoD Workshop on Evolvable Hardware*, July 2001.
13. R. B. Porter, M. Gokhale, N. R. Harvey, S. J. Perkins, and C. Young, "Evolving a spatio-spectral network on reconfigurable computing for multispectral feature identification," *Proc. SPIE* **4480**, 2001. (in this volume).
14. See `http://www.daps.lanl.gov/genie` for a description of the GENetic Imagery Exploitation (GENIE) system, as well as a list of references.
15. J. R. Hauser and J. Wawrzynek, "GARP: A MIPS processor with a reconfigurable coprocessor," in *IEEE Symposium on FPGAs for Custom Computing Machines*, J. Arnold and K. L. Pocek, eds., pp. 12–21, IEEE Computer Society Press, 1997.
16. C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, "The Napa Adaptive Processing Architecture," in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, eds., pp. 28–37, IEEE Computer Society Press, 1998.
17. M. B. Gokhale and J. M. Stone, "Co-synthesis to a hybrid RISC/FPGA architecture," *Journal of VLSI Signal Processing Systems* **24**, pp. 165–180, 2000.
18. Altera Corporation. `http://www.altera.com/products/devices/excalibur/exc-index.html`.
19. Xilinx, Inc. `http://www.xilinx.com/prs_rls/ibmpartner.htm`.
20. Quicksilver Technology, Inc. `http://www.quicksilvertech.com`.
21. Triscend Corporation. `http://www.triscend.com`.
22. Adaptive Silicon, Inc. `http://www.adaptivesilicon.com`.
23. Integrated Circuit Technology Corporation. `http://www.ictpld.com`.
24. Infinite Technology Corporation. `http://www.itc-usa.com`.
25. LightSpeed Semiconductor. `http://www.lightspeed.com`.
26. MorphICs Technology, Inc. `http://www.morphics.com`.
27. Systolix. `http://www.systolix.co.uk/nf/index.htm`.
28. M. Gokhale, J. Frigo, K. McCabe, J. Theiler, and D. Lavenier, "Early experience with a hybrid processor: k-means clustering," in *ERSA '01: First International Conference on Engineering of Reconfigurable Systems and Algorithms (Las Vegas, NV, 25-28 June 2001)*, 2000.
29. E. Forgy, "Cluster analysis of multivariate data: efficiency versus interpretability of classifications," *Biometrics* **21**, p. 768, 1965. (Abstract).
30. G. H. Bull and D. J. Hall, "ISODATA – a novel method of data analysis and pattern classification," tech. rep., Stanford Research Institute, Menlo Park, CA, 1965.
31. J. MacQueen, "On convergence of k-means and partitions with minimum average variance," *Ann. Math. Statist.* **36**, p. 1084, 1965. (Abstract).
32. J. MacQueen, "Some methods of classification and analysis of multivariate observations," in *Proceedings of Fifth Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, eds., pp. 281–297, University of California Press, (Berkeley), 1967.

33. S. P. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Information Theory* **IT-28**, pp. 129–137, 1982. This article represents the first published appearance of Lloyd's 1957 manuscript, written in the Mathematical Research Department at Bell Laboratories.
34. L. M. Ni and A. K. Jain, "A VLSI systolic architecture for pattern clustering," *IEEE Trans. on Pattern Anal. and Mach. Intel. PAMI* **7**, pp. 80–89, 1985.
35. A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *J. Roy. Stat. Soc. B* **39**, pp. 1–38, 1977.
36. J. A. Hartigan, *Clustering Algorithms*, John Wiley & Sons, New York, 1975.
37. H. Späth, *Cluster Analysis Algorithms for Data Reduction and Classification of Objects*, John Wiley & Sons, New York, 1975.
38. R. M. Gray, "Vector quantization," *IEEE ASSP Mag.* **1**, pp. 4–29, 1984.
39. H. Späth, *Cluster Dissection and Analysis: Theory, FORTRAN programs, Examples*, Ellis Horwood Limited, Chichester, UK, 1985.
40. A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data*, Prentice Hall, Englewood Cliffs, NJ, 1988.
41. A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A review," *ACM Computing Surveys* **31**, pp. 264–323, 1999.
42. P. M. Kelly and J. M. White, "Preprocessing remotely-sensed data for efficient analysis and classification," *Proc. SPIE* **1963**, pp. 24–30, 1993.
43. R. A. Schowengerdt, *Techniques for Image Processing and Classification in Remote Sensing*, Academic Press, Orlando, 1983.
44. D. M. Titterington, "Updating a diagnostic system using unconfirmed cases," *Appl. Statist.* **25**, pp. 238–247, 1976.
45. T. J. O'Neil, "Normal discrimination with unclassified observations," *J. Amer. Statist. Assoc.* **73**, pp. 821–826, 1978.
46. G. J. McLachlan and S. Ganesalingam, "Updating the discriminant function on the basis of unclassified data," *Communications in Statistics – Simulation and Computation* **11**, pp. 753–767, 1982.
47. V. Castelli and T. M. Cover, "On the exponential value of labeled samples," *Pattern Recognition Lett.* **16**, pp. 105–111, 1995.
48. P.-F. Hsieh and D. Landgrebe, "Statistics enhancement in hyperspectral data analysis using spectral-spatial labeling, the EM algorithm, and the leave-one-out covariance estimator," *Proc. SPIE* **3438**, pp. 183–190, 1999.
49. A. Blum and S. Chawla, "Learning from labeled and unlabeled data using graph minicuts," in *Proceedings of the 18th International Conference on Machine Learning*, C. E. Brodley and A. P. Danyluk, eds., pp. 19–26, Morgan Kaufmann, (San Francisco), 2001.
50. D. G. Sheppard, A. Bilgin, M. S. Nadar, B. R. Hunt, and M. W. Marcellin, "Vector quantizer for image restoration," *IEEE Trans. Image Processing* **7**, pp. 119–124, 1998.
51. C. Funk, J. Theiler, D. A. Roberts, and C. C. Borel, "Clustering to improve matched-filter detection of weak gas plumes in hyperspectral imagery," *IEEE Trans. Geosci. Remote Sensing* , 2001. In press.
52. A. D. Stocker and A. P. Schaum, "Application of stochastic mixing models to hyperspectral detection problems," *Proc. SPIE* **3071**, pp. 47–60, 1997.
53. M. E. Leeser, J. Theiler, M. Estlick, N. V. Kitaryeva, and J. J. Szymanski, "Effect of data truncation in an implementation of pixel clustering on a custom computing machine," *Proc. SPIE* **4212**, pp. 80–89, 2000.
54. NASA, 1999. `http://makalu.jpl.nasa.gov/avaris.html`.
55. P. G. Weber, B. C. Brock, A. J. Garrett, B. W. Smith, C. C. Borel, W. B. Clodius, S. C. Bender, R. R. Kay, and M. L. Decker, "MTI Mission Overview," *Proc. SPIE* **3753**, pp. 340–346, 1999.
56. J. Theiler, M. Leeser, M. Estlick, and J. J. Szymanski, "Design issues for hardware implementation of an algorithm for segmenting hyperspectral imagery," *Proc. SPIE* **4132**, 2000.
57. M. Leeser, J. Theiler, M. Estlick, and J. J. Szymanski, "Design tradeoffs in a hardware implementation of the k-means clustering algorithm.," in *Proc. SAM 2000: First IEEE Sensor Array and Multichannel Signal Processing Workshop, 16-17 March 2000, Cambridge, MA.*, pp. 520–524, 2000.
58. M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transforms in the implementation of k-means clustering on reconfigurable hardware," in *FPGA 2001: Ninth International Symposium on Field Programmable Gate Arrays*, pp. 103–110, Association for Computing Machinery, 2001.

59. M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. J. Szymanski, and J. Theiler, "Applying reconfigurable hardware to the analysis of multispectral and hyperspectral imagery," *Proc. SPIE* **4480**, 2001.
60. V. Faber, "Clustering and the continuous k-means algorithm," *Los Alamos Science* **22**, pp. 138–144, 1994.
61. M. Leeser, P. Belanovic, M. Estlick, M. Gokhale, J. J. Szymanski, and J. Theiler, "Parameterized k-means clustering for rapid hardware development to accelerate analysis of satellite data," in *High Performance Embedded Computing (HPEC) Workshop, MIT Lincoln Laboratory*, 2001.
62. P. Domingos and G. Hulten, "A general method for scaling up machine learning algorithms and its application to clustering," in *Proceedings of the 18th International Conference on Machine Learning*, C. E. Brodley and A. P. Danyluk, eds., pp. 106–113, Morgan Kaufmann, (San Francisco), 2001.
63. T. Kanungu, D. M. Mount, N. Netanyahu, C. Piatko, R. Silverman, and A. Y. Wu, "The analysis of a simple k-means clustering algorithm," *Proc. 16th ACM Symp. on Computational Geometry* , pp. 101–109, 2000. An updated version, entitled "An Efficient k-Means Clustering Algorithm: Analysis and Implementation" is available at `http:///www.cs.umd.edu/∼mount/pubs.html`.
64. J. Theiler and G. Gisler, "A contiguity-enhanced k-means clustering algorithm for unsupervised multispectral image segmentation," *Proc. SPIE* **3159**, pp. 108–118, 1997.
65. Y. Linde, A. Buzo, and R. M. Gray, "An algorithm for vector quantizer design," *IEEE Trans. Communications* **COM-28**, pp. 84–95, 1980.
66. P. Fränti, T. Kaukoranta, and O. Nevalainen, "On the splitting method for VQ codebook generation," *Opt. Eng.* **36**, pp. 3043–3051, 1997.
67. T. Kaukoranta, P. Fränti, and O. Nevalainen, "Iterative split-and-merge algorithm for vector quantization codebook generation," *Opt. Eng.* **37**, pp. 2726–2732, 1998.
68. J. W. Boardman, "Automating spectral unmixing of AVIRIS data using convex geometry concepts," in *Summaries of the Fourth Annual JPL Airborne Geoscience Workshop*, R. O. Green, ed., pp. 11–14, 1994.
69. J. W. Boardman, "Geometric mixture analysis of imaging spectrometry data," *Proc. IGARSS (IEEE International Geoscience and Remote Sensing Symposium)* , pp. 2369–2371, 1994.
70. A. Cutler and L. Breiman, "Archetypal analysis," *Technometrics* **36**, pp. 338–347, 1994.
71. J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping target signatures via partial unmixing of AVIRIS data," in *Summaries of Fifth Annual JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 23–26, 1995.
72. D. A. Roberts, M. Gardner, R. Church, S. Ustin, G. Scheer, and R. O. Green, "Mapping chaparral in the Santa Monica mountains using multiple spectral mixture models," in *Summaries of 6th JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 197–201, 1996.
73. S. Tompkins, J. F. Mustard, C. M. Pieters, and D. W. Forsyth, "Optimization of endmembers for spectral mixture analysis," *Remote Sensing of the Environment* **59**, pp. 472–489, 1997.
74. G. S. Okin, W. J. Okin, D. A. Roberts, and B. Murray, "Multiple endmember spectral mixture analysis: Application to an arid/semi-arid landscape," in *Summaries of the Seventh JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 291–299, 1998.
75. C. A. Bateson, G. P. Asner, and C. A. Wessman, "Incorporating endmember variability into spectral mixture analysis through endmember bundles," in *Summaries of the Seventh JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 43–52, 1998.
76. J. Bowles, M. Daniel, J. Grossman, J. Antoniades, M. Baumback, and P. Palmadesso, "Comparison of output from ORASIS and pixel purity calculations," *Proc. SPIE* **3438**, pp. 148–156, 1998.
77. A. Mooijaart, P. G. M. van der Heijden, and L. A. van der Ark, "A least squares algorithm for a mixture model for compositional data," *Computational Statistics and Data Analysis* **30**, pp. 359–379, 1999.
78. M. E. Winter, "N-FINDR: an algorithm for fast autonomous spectral end-member determination in hyperspectral data," *Proc. SPIE* **3753**, pp. 266–277, 1999.
79. D. D. Lavenier, J. Theiler, J. J. Szymanski, M. Gokhale, and J. R. Frigo, "FPGA implementation of the pixel purity index algorithm," *Proc. SPIE* **4212**, 2000.
80. J. Theiler, D. D. Lavenier, N. R. Harvey, S. J. Perkins, and J. J. Szymanski, "Using blocks of skewers for faster computation of pixel purity," *Proc. SPIE* **4132**, pp. 61–71, 2000.

Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512

Tim Grembowski[1], Roar Lien[1], Kris Gaj[1], Nghi Nguyen[1],
Peter Bellows[2], Jaroslav Flidr[2], Tom Lehman[2], Brian Schott[2]

[1]Electrical and Computer Engineering, George Mason University, 4400 University Drive, Fairfax, VA 22030
{rlien, kgaj, nnguyen1}@gmu.edu
[2] University of Southern California - Information Sciences Institute
Arlington, VA 22203
{pbellows, jflidr, tlehman, bschott}@east.isi.edu

Abstract. Hash functions are among the most widespread cryptographic primitives, and are currently used in multiple cryptographic schemes and security protocols such as IPSec and SSL. In this paper, we compare and contrast hardware implementations of the newly proposed draft hash standard SHA-512, and the old standard, SHA-1. In our implementation based on Xilinx Virtex FPGAs, the throughput of SHA-512 is equal to 670 Mbit/s, compared to 530 Mbit/s for SHA-1. Our analysis shows that the newly proposed hash standard is not only orders of magnitude more secure, but also significantly faster than the old standard. The basic iterative architectures of both hash functions are faster than the basic iterative architectures of symmetric-key ciphers with equivalent security.

1 Introduction

Hash functions are very common and important cryptographic primitives. Their primary application is their use together with public-key cryptosystems in the digital signature schemes. They are also a basic building block of secret-key Message Authentication Codes (MACs), including the American federal standard HMAC [8]. This authentication scheme appears in two currently most widely deployed security protocols, SSL and IPSec [12, 16]. Other popular applications of hash functions include fast encryption, password storage and verification, computer virus detection, pseudo-random number generation, and many others [13, 16].

Cryptographically strong, collision-free, hash functions are very difficult to design. Tens of them have been proposed, and the majority of them have been broken. Only a few hash functions have gained a wider acceptance, and even fewer have been standardized.

By far the most widely accepted hash function is SHA-1 (Secure Hash Algorithm-1), a revised version of the American federal standard introduced in 1993 [4]. The original version of this function, SHA, was developed by National Security Agency

(NSA), and revised in 1995 for increased security even before any weakness was found in the open research.

SHA-1 was introduced as a federal standard about the same time as an 80-bit secret-key encryption algorithm named Skipjack [5] and the Digital Signature Standard (DSS) [6]. The security parameters of all these standards were chosen in such a way to guarantee the similar level of security, in the range of 2^{80} operations, as required by the best currently known attack.

After introducing a new secret-key encryption standard, AES (Advanced Encryption Standard) [7], with three key sizes, 128, 192, and 256 bits, the security of SHA-1 does not any longer match the security guaranteed by the encryption standard. Therefore, an effort was initiated by NSA to develop three new hash functions, with the security matching the security of AES with 128, 192, and 256 bit key respectively. This effort resulted in the publication of the draft Federal Information Processing Standard, introducing three new hash functions referred to as SHA-256, SHA-384, and SHA-512 [11].

The goal of the project described in this article was to implement the most complex of these new hash functions, SHA-512, in reconfigurable hardware, and to compare its implementation with the implementation of SHA-1, realized in the same technology.
Our comparative analysis sought, among the other, answers to the following questions:

- does the increased security of the SHA-512 hash function come at the cost of decreased speed, increased area, or decreased speed to area ratio of the hardware implementations when compared to the SHA-1 hash function;
- how does the speed of the SHA-512 hash function compare to the speed of the corresponding versions of the AES algorithm? Which transformation, encryption or authentication, is faster in hardware? Which transformation requires less area?

Our investigation is a part of the larger project [10] aimed at implementing a hardware accelerator for a new suite of cryptographic algorithms to be used in the IP security protocol, IPSec. The target throughput of this accelerator is 1 Gbit/s for both encryption and authentication. Therefore we are also interested in studying the difficulty of implementing SHA-1 and the newly proposed hash functions at the speed of 1 Gbit/s using the current FPGA devices.

Although multiple commercial and academic implementations of SHA-1 have been reported and validated by NIST [15], we are not aware of any hardware implementation of SHA-512, or its comparison with the implementation of SHA-1 implemented in the same technology, using the same optimization techniques. This article is aimed at filling this gap.

2 Functional Comparison

In Table 1, four investigated hash functions are compared from the point of view of functional characteristics. The security of these hash functions is determined by the size of their outputs, referred to as hash values, *n*. The best known attack against these functions, the “birthday attack”, can find a pair of messages having the same hash value with a work factor of approximately $2^{n/2}$. This complexity means that in order to accomplish equivalent security, hash functions need to have an output twice as long as the size of a key of the corresponding secret-key cipher.

SHA-1 and SHA-256 have many features in common. They both can process messages with the maximum length up to 2^{64}-1 bits, have a message block size of 512 bits, and have internal structure based on processing 32-bit words. SHA-384 and SHA-512 have even more similarities. They process messages with the maximum length up to 2^{128}-1 bits, have a message block size of 1024 bits, and have internal structure based on processing 64-bit words. On top of that, the definition of SHA-384 is almost identical to the definition of SHA-512, with the exception of a different choice of the initialization vector, and a truncation of the final 512-bit result to 384 bits.

All functions have a very similar internal structure, and process each message block using multiple rounds. The number of rounds is the same for SHA-1, SHA-384, and SHA-512, and 20% smaller in SHA-256. The critical path in each round involves multioperand addition. SHA-1 requires two fewer operands per addition than in the remaining three functions.

A notation k+1 used in the table, means that the number of operands to be added is k in all but last round, and k+1 in the last round. Alternatively, a number of operands may be equal to k in all rounds, and an additional simplified round may be introduced for the remaining single addition.

Table 1. Functional characteristics of four investigated hash functions

	SHA-1	**SHA-256**	**SHA-384**	**SHA-512**
Size of hash value	160	256	384	512
Complexity of the best attack	2^{80}	2^{128}	2^{192}	2^{256}
Equivalently secure secret-key cipher	Skipjack	AES-128	AES-192	AES-256
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Message block size	512	512	1024	1024
Word size	32	32	64	64
Number of words	5	8	8	8
Number of digest rounds	80	64	80	80
Number of operands added in the critical path	5+1	7+1	7+1	7+1
Number of constants K_t	4	64	80	80
Round-dependent operations	f_t	None	None	None

The number of different constants is equal to four in SHA-1, and is the same as the number of rounds in all remaining functions. As a result, implementations of SHA-256, SHA-384, and SHA-512 must include a look-up table of constants, K_t, where t=0..number of rounds. SHA-1 is also the only function that contains an operation dependent on the round number t; in all remaining hash functions all rounds perform exactly the same operations.

The following conclusions can be derived from this functional comparison. Hardware implementations of SHA-384 and SHA-512 have exactly the same performance, so only one of them needs to be implemented for the purpose of comparative analysis. Notice that the size of the message block is twice as large in SHA-512 as compared to SHA-1, the number of rounds is the same, and the critical path is only slightly longer in SHA-512. Because of this, SHA-512 (the strongest function) is likely to be significantly faster than SHA-1 (the weakest function), which would be a very positive result if true. The throughput of SHA-256 is likely to be in the same range as a throughput of SHA-1, and smaller than the throughput of SHA-512. Taking into account these estimations, we have decided to implement two of the investigated hash functions, SHA-1 and SHA-512, which lay on the opposite ends of the spectrum in terms of both security and speed, with SHA-1 being the weakest and slowest, and SHA-512 being the strongest and fastest of the four investigated hash functions.

3 Design Methodology

Our target FPGA device was the Xilinx Virtex XCV-1000-6. This device is composed of 12,288 basic logic cells referred to as CLB (Configurable Logic Block) slices, includes 32 4-kbit blocks of synchronous dual-ported RAM, and can achieve synchronous system clock rates up to 200 MHz [17]. This device was chosen because of the availability of a general purpose PCI board based on three FPGA devices of this type. This board is described in detail in Section 5.

The design flow and tools used in our group for the implementation of cryptographic modules in Xilinx FPGA devices are shown in Fig. 1. All algorithms were first described in VHDL, and their description verified through the functional simulation using Active HDL v. 5.1, from Aldec, Inc. Test vectors and intermediate results from the reference software implementations based on the Crypto++ library [1] were used for debugging and verification of VHDL codes. The revised VHDL code became an input to the Xilinx integrated environment ISE 4.1i, performing the automated logic synthesis, mapping, placing, and routing. Tools included in this environment generated reports describing the area and speed of implementation, a netlist used for timing simulation, and a bitstream used to configure an actual FPGA device. All designs were fully verified through behavioral, post-synthesis, and timing simulations, and experimentally tested using the procedure described in Section 5.

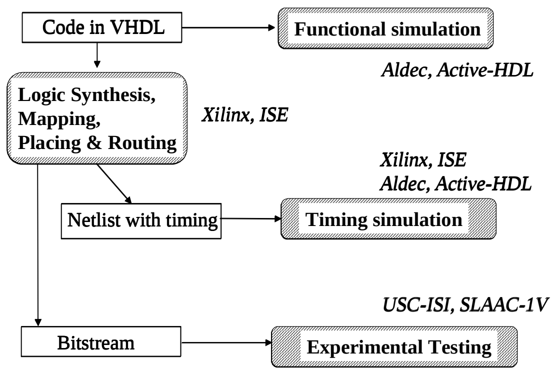


Fig. 1. Design flow and tools used in the development of cryptographic modules

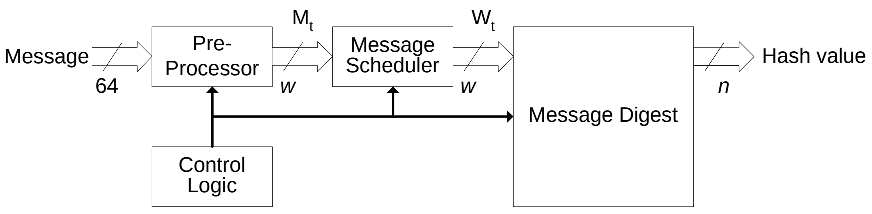


Fig. 2. General block diagram of SHA-1 and SHA-512. For SHA-1, w=32, n=160; for SHA-512, w=64, n=512

4 Hardware Architectures

A general block diagram common for all four hash functions is shown in Fig. 2. Input messages pass first through the preprocessing unit which performs padding and forms message blocks of the fixed length, 512 or 1024 bits, depending on the hash function. The preprocessing unit passes message blocks to the message scheduler unit. In our architecture, message blocks are passed to the message scheduler unit a word at a time, during the first 16 clock cycles used to process each message block. The message digest unit performs the actual hashing. It uses one clock cycle per digest round. In each round, the digest unit processes a new word W_t generated by the message scheduler unit.

The internal structure of the message digests for SHA-1 and SHA-512 are shown in Fig. 3ab. In both functions, input registers are initialized with the constant initialization vector, and are updated with the new value in each round. In SHA-1, four out of five words (A, B, C, and D) remain almost unchanged by a single round. These words are only shifted by one position down. The last word, E, undergoes a complicated transformation equivalent to multioperand addition modulo 2^{32}, with five 32-bit oper-

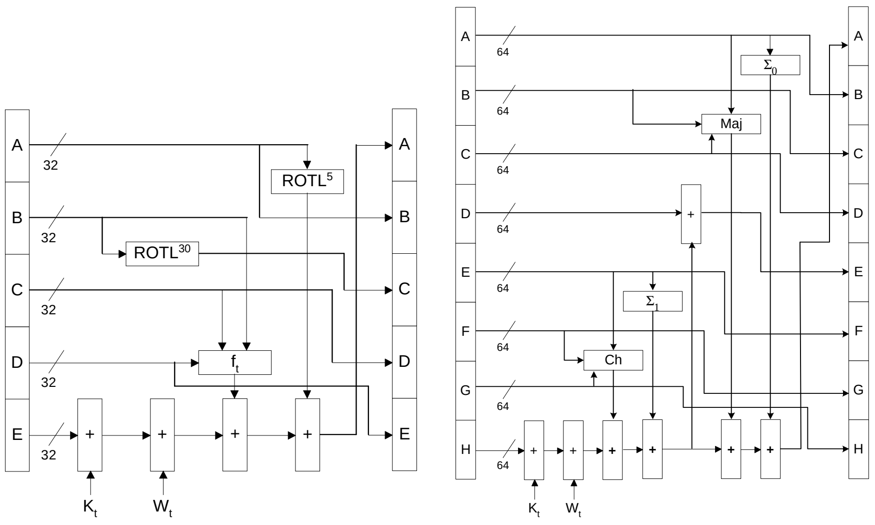


Fig. 3. Functional block diagram of the message digest unit of a) SHA-1, b) SHA-512

ands dependent on all input words, the round-dependent constant K_t, and the message dependent word W_t. The internal structure of the message digest of SHA-512 is similar. The primary differences are as follows: The number of words processed by each round is 8, each word is 64 bits long, and the longest path is equivalent to addition of seven 64-bit operands modulo 2^{64}. These operands depend on seven out of eight input words (all except D), the round-dependent constant K_t, and a message dependent word W_t. Six out of eight input words remain unchanged by a single round.

Our implementations of the message digests are shown in Figs. 4ab. The critical path in each circuit is marked with a thick line. Both circuits use the carry save representation of numbers to speed-up the multioperand addition, and minimize delays associated with carry propagation. The number of operands that need to be processed in each round has been minimized by precomputing the sum $K_t + W_t$ in the preceding clock cycle.

At the same time, the need for an additional round at the end of processing has been eliminated by introducing a conditional addition of the initial value of registers H_0-H_m (m=4 for SHA1, and m=7 for SHA-512) inside of each round. These initial values are added only in the last round of the message digest computations; in all previous rounds zero is added instead. After these two optimizations, the maximum number of operands to be added in each round is 5 for SHA-1 and 7 for SHA-512.

The straightforward use of carry save adders in case of five operand addition would lead to three levels of 3-to-2 carry save adders, followed by a carry propagate adder as shown in Fig. 5a. Instead, we have decided to use a 5-to-3 parallel counter (see Fig. 5b) [14], which reduces the number of binary digits at each position in the sum of five operands from 5 to 3, and has approximately the same delay as a 3-to-2 carry save adder. The operation of the 5-to-3 parallel counter is shown in Fig. 5c, using the dot notation. In this notation, each dot represents a binary digit, 0 or 1 [14]. The 5-to-3

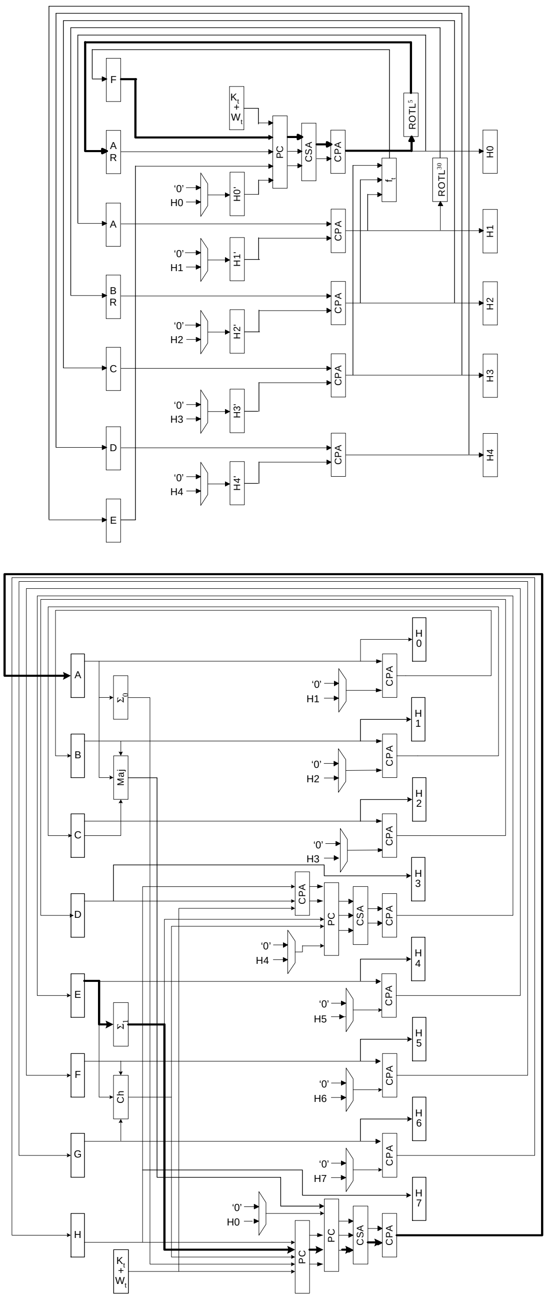


Fig. 4. Our implementations of the message digest units of a) SHA-1, b) SHA-512

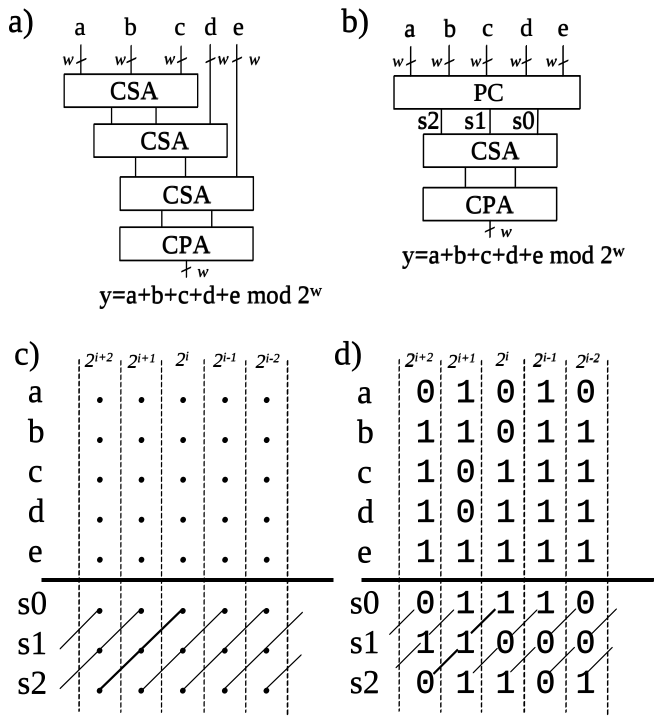


Fig. 5. Using 5-to-3 Parallel Counter. a) adding five *w*-bit numbers using a tree of 3-to-2 carry-save adders, b) adding five *w*-bit numbers using 5-to-3 parallel counter followed by a 3-to-2 carry save adder, c) operation of the 5-to-3 parallel counter in the dot notation, d) example of the operation of the 5-to-3 parallel counter

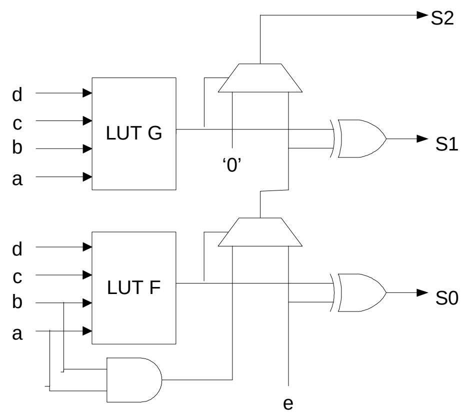


Fig. 6. Using internal structure of a single CLB slice of the Xilinx Virtex FPGA device to implement a bit-slice of a 5-to-3 Parallel Counter (PC)

parallel counter adds five binary digits with the same weight, 2^i, and represents the result using three binary digits with three subsequent weights, 2^i, 2^{i+1}, and 2^{i+2}. An example of the operation of this counter is shown in Fig. 5d. The speed-up comes from the fact that the operation of the parallel counter can be realized in Virtex FPGAs using resources of a single CLB slice as shown in Fig. 6.

In SHA-512, a cascade of two 5-to-3 parallel counters is used to reduce the number of operands from seven to three (see Fig. 4b). As a result, the critical path is longer than in SHA-1 only by two levels of CLB slices (one level for the parallel counter, and one for the $\sum_1$ operation).

Further optimization of the critical path in both circuits has been accomplished by reducing the delays of interconnects. The primary optimization technique used for that purpose was the reduction of the fan-out of control signals by using buffers, duplicating portions of control logic, and placing control logic close to the controlled parts of the execution unit.

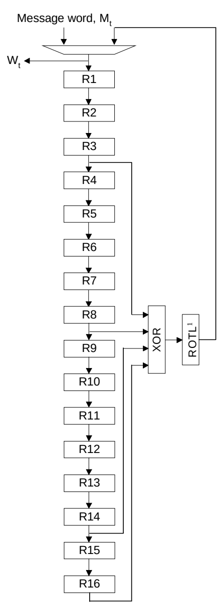
The block diagrams of the message scheduling units in SHA-1 and SHA-512 are shown in Fig. 7. Both units generate 80 message dependent words, W_t, t=0..79. The first 16 of these words, $W_0..W_{15}$, is simply the first 16 words of the input message block, $M_0..M_{15}$; the remaining words are computed using a simple feedback function, based on rotations, shifts, and XOR operations. The actual implementation of both functions is given in Fig. 8. Our implementations have been optimized for minimum area, using a shift register mode of CLB slices available in the Xilinx Virtex FPGA devices. Using this mode, a cascade of several one-bit registers, each taking normally a single CLB slice, can be reduced to a single CLB slice implementing the multi-stage shift register with up to 16 stages.

5 Testing Procedure

The experimental testing of our cryptographic modules was performed using the SLAAC-1V hardware accelerator board. The logical architecture of SLAAC-1V is shown in Fig. 9. The three Virtex 1000 FPGAs (denoted as X0, X1, and X2) are the primary processing elements.

About 20% of the resources in the X0 FPGA are devoted to the PCI interface and the board control module. The remaining logic of this device, as well as the entire X1 and X2 FPGAs, can be used by the application developer. The board control module implemented in X0 provides high-speed DMA (Direct Memory Access), data buffering, clock control (including single-stepping and frequency synthesis from 1 to 200 MHz), etc. The current 32 bit 33 MHz control module has obtained DMA transfer rates of over 1 Gbit/s (125 MB/s) between X0 and the host memory, very near the PCI theoretical maximum.

In all our experiments, the X1 FPGA was configured to contain cryptographic modules, while X0 and X2 were used only to facilitate the transfer of data between X1 and the memory of the host computer running Linux.



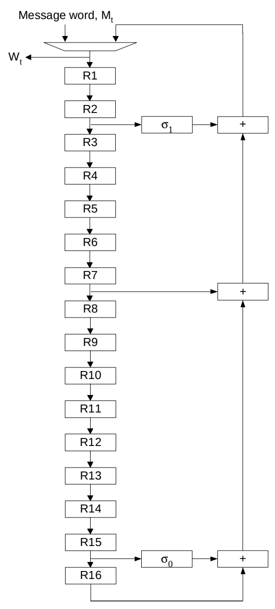
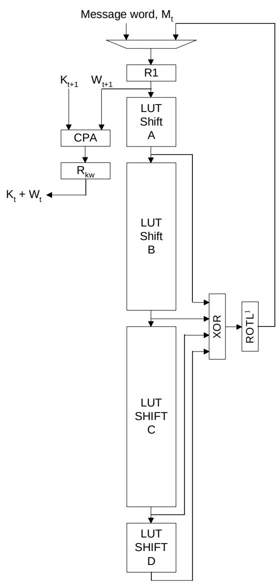


Fig. 7. Functional block diagrams of the message scheduler unit of a) SHA-1, b) SHA-512



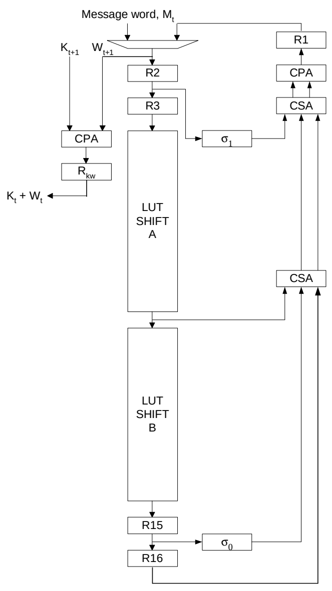


Fig. 8. Our implementations of the message scheduler unit of a) SHA-1, b) SHA-512

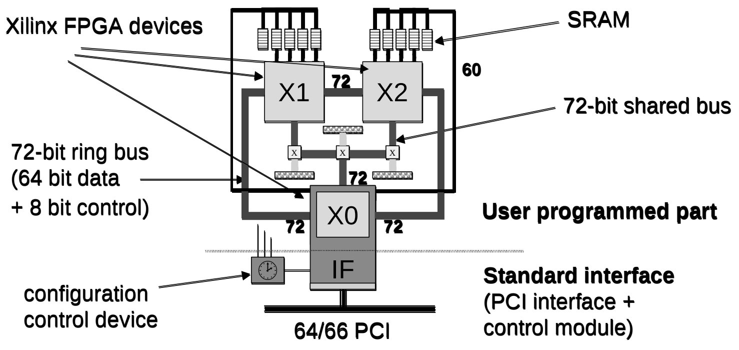


Fig. 9. SLAAC-1V Architecture

The test program written in used the SLAAC-1V APIs and the SLAAC-1V driver to communicate with the board.

Our testing procedure is composed of three groups of tests. The first group attempts to verify the circuit functionality at a single clock frequency. The goal of the second group is to determine the maximum clock frequency at which the circuit operates correctly. Finally, the purpose of the third group is to determine the limit on the maximum encryption and decryption throughput, taking into account the limitations of the PCI interface.

Our first group of tests is based on the NIST recommendations provided in [2]. These recommendations describe the comprehensive suite of three functional tests for SHA-1.

The second test is aimed at determining the maximum clock frequency of the hash function modules. Three megabytes of pseudorandomly generated data are sent to the board for hashing, the result is transferred back to the host and compared with the corresponding output obtained using software implementation of the given hash function based on the Crypto++ library [1]. This procedure is repeated 30 times using the same clock frequency to minimize the effect of input data values on the results of analysis. The next clock frequency is chosen based on the rules of the binary search, i.e., in the middle between two closest earlier identified frequencies giving different test results. The test is repeated until the difference between these two frequencies is smaller than the required accuracy of the measurement (< 0.1 MHz in our tests). The highest investigated clock frequency at which no single processing error is detected is considered the maximum clock frequency. In our experiments, this test was automatically repeated 10 times with consistent results in all iterations.

The third group of tests is an extension of the second group. After determining the maximum clock frequency, we measure multiple times and average the amount of time necessary to process 3 MB of data, taking into account the delay contribution of the 32 bit/33 MHz PCI interface.

6 Results

In Fig. 10, the minimum clock periods of SHA-1 and SHA-512 obtained using static timing analysis and experiment are given. For clock periods determined through static timing analysis, the percentage of the critical path delay used by logic and routing respectively is shown.

Based on the knowledge of the minimum clock period, the maximum data throughput has been computed according to the equation:

*Throughput = Message_block_size / (Clock period * Number_of_rounds)*

Throughput values calculated based on the minimum clock periods obtained using static timing analysis and experiment are shown in Fig. 11. In the same figure, these

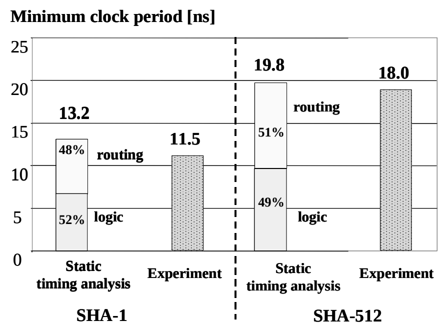


Fig. 10. Minimum clock period of SHA-1 and SHA-512: a) obtained using static timing analysis, b) determined experimentally

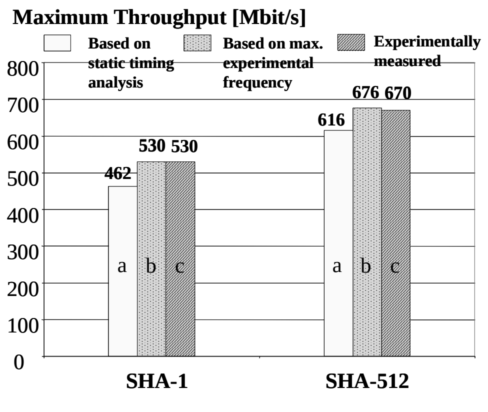


Fig. 11. Maximum throughputs of SHA-1 and SHA-512: a) obtained using static timing analysis, b) calculated based on the experimentally measured maximum clock frequency, c) experimentally measured, including the contributions of the PCI interface

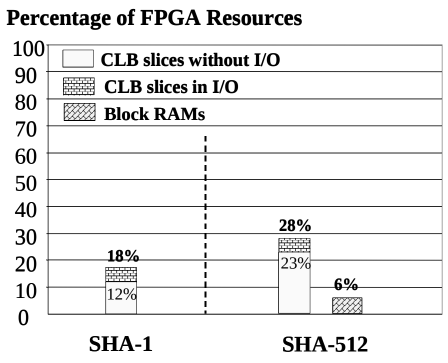


Fig. 12. Percentage of the FPGA resources used by each implementation

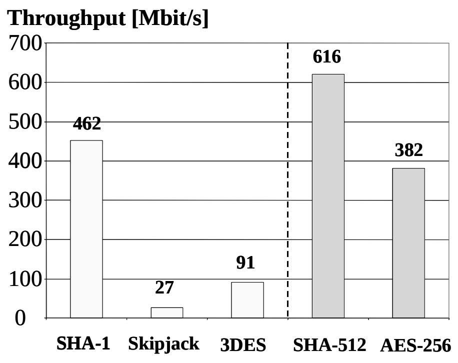


Fig. 13. Comparison of throughputs for the basic iterative architectures of old and new standards in the area of hash functions and symmetric-key ciphers

results are compared with the experimentally measured data throughputs that take into account the delay contributions of the PCI interface. This comparison demonstrates that the PCI interface provides a constant uninterrupted flow of data and has a negligible influence on the data throughput.

Our results confirm our earlier predictions that the design of strong hash functions does not involve any major trade-off between security and performance. To the contrary, the most secure function, SHA-512, is also the fastest of four investigated hash functions.

The percentage of the FPGA resources (CLB slices and Block RAMs) used by implementations of SHA-1 and SHA-512, are shown in Fig. 12. The difference in the number of CLB slices is primarily caused by the difference in the size of input and output registers in the message digest units of both functions (512 bits vs. 160 bits), and the width of the multioperand adders in the critical path of these units (64 bits vs. 32 bits). In SHA-512, two 4 kbit block RAMs are used to store 80 64-bit constants K_t.

7 Possible Extensions

The analysis of our results, reveals a potential for further optimizations. Since a large percentage of the critical path delay (48% in SHA-1 and 51% in SHA-512) is contributed by delays of interconnects, a substantial gain can be accomplished by manual floorplaning and routing. Since these optimizations are specific for a given type of the device, and are not easily transferable to another family of FPGA devices, they have not been attempted at this point.

A further radical improvement of the circuit speed can be achieved by using an unrolled architecture of the message digest unit, in which *m* (*m*=2, 4, 5, or 8) digest rounds are implemented as combinational logic and executed in the same clock cycle. As a result, the total number of clock cycles necessary to compute a digest for a single message block is reduced by a factor of *m*, at the cost of a significantly smaller increase in the delay of the critical path. The preliminary study of this extended architecture for SHA-1 and *m*=5 indicates that the delay of logic in the critical path increases by a factor smaller than 2.5, leading to the overall increase in the circuit throughput by a factor of 2. This preliminary result needs to be verified, taking into account the delays of interconnects, and will be reported in a future article together with results of a similar optimization of SHA-512.

Another popular way of speeding up the hardware implementations of cryptographic transformations is parallel processing, using either several independent execution units or pipelining. Even taking into account limitations imposed by the area of FPGA devices, pipelining was shown to permit speeding up the implementations of AES and other secret key ciphers by at least an order of magnitude [3, 9]. Taking into account the relatively smaller area required by the basic implementations of SHA-1 and SHA-512, a potential speed-up is even greater in case of hash functions.

Unfortunately, applying parallel processing to hash functions is limited by the fact that only input blocks belonging to *different* messages may be processed in parallel. In this respect, hashing is similar to encryption in the CBC (Cipher Block Chaining) mode and other *feedback* modes of secret-key block ciphers. The processing of the next message block cannot start before the processing of the previous block is fully completed. Additionally, hash functions do not possess any non-feedback modes of operation, such as ECB (Electronic CodeBook) or counter mode. Therefore, pipelining, although possible, is limited by the availability of multiple independent messages that could be processed in parallel. This availability is application specific and may strongly depend on the characteristic of the network traffic, e.g., an average size of packets exchanged in the given communication protocol.

8 Summary

An FPGA implementation of the newly proposed draft hash standard SHA-512 has been developed and compared with the implementation of the old hash standard SHA-1. An effort was made to use exactly the same technology and identical design and optimization techniques. Our implementations based on Xilinx XCV-1000-6 demonstrate that SHA-512 is 33% faster than the equivalent implementation of SHA-1 according to the static timing analysis, and 26% faster according to the experiment. At the same time, without taking into account an input/output interface, SHA-512 takes almost twice as many CLB slices as SHA-1, and requires two additional 4 kbit Block RAMs. These results have been verified experimentally using the PCI FPGA Board, SLAAC-1V, based on three Xilinx FPGA devices Virtex 1000. Our results prove that the design of a strong hash function does not necessarily involve a trade-off between the hardware speed and cryptographic security. At the same time, the more secure hash function may require substantially more hardware resources.

Further optimizations of both implementations based on loop unrolling are possible and will be reported in a future article. Our research is a part of a larger effort aimed at implementing the newly proposed cryptographic algorithms of IPSec in the form of a giga-bit rate hardware accelerator on a Xilinx FPGA-based PCI card [10].

References

1. Crypto++, free C++ class library of cryptographic schemes, available at http://www.eskimo.com/~weidai/cryptlib.html.
2. Digital Signature Standard Validation System (DSSVS) User's Guide available at http://csrc.nist.gov/cryptval/shs.html
3. Elbirt, A. J., Yip, W., Chetwynd, B., Paar, C.: An FPGA implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. Proc. 3rd Advanced Encryption Standard (AES) Candidate Conference, New York, April 13-14, 2000.
4. http://www.itl.nist.gov/fipspubs/fip180-1.htm
5. FIPS 185, Escrowed Encryption Standard (EES), February 1994.
6. FIPS 186-2, Digital Signature Standard (DSS), February 2000, available at http://csrc.nist.gov/encryption/tkdigsigs.html
7. NIST, FIPS Publication 197, Specification for the Advanced Encryption Standard (AES), November 26, 2001, available at http://csrc.nist.gov/encryption/aes/.
8. FIPS 198, HMAC - Keyed-Hash Message Authentication Code, available at http://csrc.nist.gov/encryption/tkmac.html
9. Gaj, K., and Chodowiec, P.: Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard Using Field Programmable Gate Arrays, Proc. RSA Security Conference - Cryptographer's Track, April 2001.
10. GRIP (Gigabit Rate IP Security) project page, available at http://www.east.isi.edu/projects/GRIP/
11. NIST Cryptographic Toolkit, Secure Hashing, available at http://csrc.nist.gov/encryption/tkhash.html
12. IP Security Protocol (ipsec) Charter - Latest RFCs and Internet Drafts for IPSec, http://ietf.org/html.charters/ipsec-charter.html
13. Menezes, A. J., van Oorschot P. C., and Vanstone S. A.: Handbook of Applied Cryptography, CRC Press, Inc., Boca Raton, 1996.
14. Parhami, B.: Computer Arithmetic: Algorithms and Hardware Design, Oxford University Press, 2000.
15. SHS Validation List, available at http://csrc.nist.gov/cryptval/shs/shaval.htm.
16. Stallings, W.: Cryptography and Network Security, 1999 Prentice-Hall, Inc., Upper Saddle River, New Jersey. 2nd Edition.
17. Xilinx, Inc.: Virtex 2.5 V Field Programmable Gate Arrays, available at www.xilinx.com.

Configurable Computing and Sonar Processing - Architectures and Implementations

Brent E. Nelson
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT, 84604
nelson@ee.byu.edu [*]

Abstract

Sonar beamforming is an ideal application for reconfigurable computing due to its high available levels of parallelism, relatively low sample rates, and modest word sizes. In this paper we describe a family of beamforming algorithms and their implementation using configurable computing technology. These include algorithms for time-delay, frequency-domain, and matched field beamforming. Configurable computing architectures appropriate for each are described and the tradeoffs associated with the mapping of each to concrete platforms is discussed.

1 Introduction

FPGA technology provides attractive solutions for a range of applications areas. With their inherent reprogrammability, FPGA's exhibit characteristics normally associated with programmable processors. At the same time, they often provide solutions with order-of-magnitude performance advantages over programmable processors.

While the various vendors' FPGA offerings differ in many details, the basics are the same, consisting of an array of logic blocks and wires. A number of innovations introduced by the vendors over time have made it possible to push more and more of the system design onto the FPGA. For example, many vendors offer fixed-function blocks inside the FPGA chip. These include wide decoders, shift registers, digital phase lock loops, embedded memories, and multipliers. The latest devices contain hundreds of such blocks distributed throughout the FPGA chip.

Like all semiconductor products, FPGA's have historically followed Moore's Law where the number of components on a chip is doubling every 18 months. Today's devices have advertised gate capacities in the millions. Until recently, it was difficult to contemplate the use of FPGA's for many DSP computations due to the need for wide-word arithmetic and floating point computations. However, as FPGA's continue to grow in density these are now feasible — while FPGA's are not *catching up* with ASIC's in terms of raw performance, they have crossed the density threshold required for use in many advanced DSP applications. As such, factors like time-to-market and NRE costs are combining to make FPGA's competitive in many cases with ASIC's for application-specific DSP solutions.

Algorithms implemented on DSP processors often have the following characteristics: (a) large amounts of exploitable parallelism, (b) modest data word sizes (on the order of 8-24 bits), and (c) relatively simple control algorithms that can often be statically scheduled. Such algorithms are also well suited to modern FPGA devices.

1.1 Parallelism

The sheer size of modern FPGA devices makes it possible to exploit much of the available parallelism, both temporally (pipelining) and spatially (parallel processing). Pipelining can often be used to obtain large speedups over sequential processing for sonar computations by creating deeply pipelined datapaths which can retire one inner loop computation every clock cycle. As will be seen later, this is possible for relatively complex inner loop computations resulting in 12 or more arithmetic operations per cycle.

Spatial parallelism is the use of multiple processing elements (PE's) in parallel to solve a problem. A key feature of parallel implementations on FPGA's is that the interprocessor communication can be very lightweight, usually consisting of simple flags to indicate the presence of valid data on wires. This commonly results in linear parallel speedups, even for large numbers of processing elements.

A limitation to using any kind of parallelism in a computation is the availability of memories for fetching operands and storing results — the limitation is often not chip area nor total memory size but rather the number of independently accessible memories available. FPGA designs

[*]Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

making use of tens or even hundreds of separate memories are feasible, further contributing to speedup via parallelism.

1.2 Arithmetic and Word Size

Increasing FPGA densities are making feasible the implementation of more arithmetically intensive algorithms as time goes on. One of the pioneering configurable computing machines developed was Splash-2, a general-purpose configurable computing machine designed in the early 1990's. Splash-2 was based on Xilinx 4010 FPGA's. A total of 50 16-bit integer additions would fit on a 4010. The comparable figure for today's largest Virtex-II device is $5,800$ 16-bit integer additions. Interestingly, the comparable figure for 16-bit floating point adder/subtractors today is in the hundreds (325 for combinational, 109 for fully pipelined). Also, work such as that in [1] illustrates the advantages to be gained by using fixed point arithmetic as well as using custom word widths at each point in a computation. This is in contrast to a processor-based solution which employs a single word size.

1.3 Hardwired Control

The simple control schemes used in many DSP algorithms can be directly implemented as fast, customized state machines of moderate complexity. The design can be organized so that the control and data-path circuitry interact in ways which allow the datapath to operate at 100% efficiency with no interference from the control. This emphasis on the design of datapaths which retire one complete inner loop computation per clock cycle is a recurring theme throughout the remainder of this paper.

1.4 Configurable Computing Machines

The work described below was completed in the context of specific Configurable Computing Machines (CCM's). A CCM is a machine, based on FPGA's which can be used as a target for a range of computations. A CCM typically sits on the bus of a host processor with the processor controlling its FPGA configuration, clocking, and I/O. When FPGA densities were low, CCM's contained many FPGA's (Splash-2 contained 17 per board and multiple boards could be used in parallel). As FPGA densities have increased, the part count on CCMs has actually decreased due to space and power considerations. The Osiris board, developed as a part of the SLAAC project (*http://www.east.isi.edu.projects/SLAAC*), contains a single Virtex-II FPGA (XC2V6000) connected to 10 SRAM's, two SODIMM memory slots, and a PCI interface controller. In addition, the XC2V6000 contains 144 internal memories of 18K bits each. The availability of so many memories will prove to be a key factor in the implementations described below.

To illustrate the above principles regarding DSP on FPGA's, the remainder of this paper evaluates the suitability of FPGA's for sonar beamforming by describing a number of mapping experiments undertaken at BYU. Each mapping experiment is used to illuminate one or more of the above points. Some of the described mappings were reduced to hardware while others are paper designs. The validity of the paper designs was established by a number of means including comparing to the completed designs while taking into account technology differences between the implemented design's technology and that used in Osiris.

2 Time Delay Beamforming

Time-delay beamforming takes advantage of the fact that the direction of arrival of a wave can be determined by measuring the delay between the times the wave strikes each sensor in an array. Steering a beam in a particular direction consists of appropriately delaying the response from each sensor in the array and summing those delayed sensor values. Because the signals of interest are periodic, signals coming from the direction of interest will coherently add while signals not coming from the direction of interest will destructively cancel. The result of the summation thus indicates the broadband energy arriving from a particular direction.

In its very simplest form, time delay beamforming can be expressed by the following pseudo-code:

```
formBeam(BeamDirection b) {
  response = 0;
  for (s=0;s<numSensors;s++)
    response = response +
            dataSamples[s][delay(b,s)];
}
```

A key component is the delay function (*delay(b,s)*) — it can be implemented as a computation (a function of array geometry and look direction) or a simple table lookup.

The computation contains much parallelism. Our first implementation will use the internal FPGA memories to hold both the delay lookup table and the sampled data. The amount of parallelism achievable in this case is limited by the number and capacity of memories available (it is not compute-bound). The inner loop consists simply of two memory lookups followed by an accumulate. The implementation consists of one PE per sensor with each PE containing a single dual-ported memory, addressing logic, and a registered adder. A beam response is computed in a linear systolic fashion with each PE contributing one sensor's data to the summation. This is shown in Figure 1. The result is 144 PE's per chip operating in parallel. By pipelining the two memory fetches from the same dual-ported BlockRAM, each PE can retire one inner loop computation per cycle. Thus, the complete chip can achieve a computational rate of 144 arithmetic operations per clock cycle.

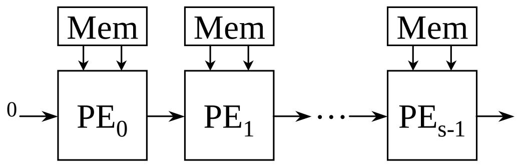


Figure 1: Time-Delay Beamformer Organization

The problem with this design is that the capacity of each on-chip memory limits the problem size for which this is applicable to about 128 sensors and $2,048$ beams. At this problem size, the PE utilization is only $1-2\%$. That is, the PE's can compute the required beams in $1-2\%$ of the available sample period and are then idle until the next samples arrive. However, the logic utilization of this design is on the order of 10% and the unused space can be used for other system circuitry.

An alternative to the above is to store the sampled data in the on-chip memories and the delay function lookup tables in the 10 off-chip memories. In this organization, 30 delay values would be loaded into the FPGA each cycle and the design would contain 30 PE's. The advantage of using off-chip storage is that it allows for the computation of much larger problems. With 30 PE's forming $6,000$ beams for $1,024$ sensors' data, the PE utilization would increase to 100% and the sustained inner loops per second would be about 6 billion when running at $150MHz$. As with the first design above, the utilization for the chip would be on the order of 10% and additional circuitry could be placed into the unused space.

A third alternative is to compute the delay function on the fly, eliminating the need for the delay table entirely. This would be done by computing the dot product of the vector representing the sensor position in a 3D coordinate space with a unit vector in the desired look direction, and dividing by the speed of signal propagation. The only storage required in each PE would be for storing the sampled data. Not only does this allow for very large problems but also has the significant advantage that dynamic modifications to account for a changing array shape can be accomodated by simply updating each PE with its sensor location in real-time. With this approach the PE area becomes the limitation. The Osiris board would support a total of 72 PE's with a computational throughput of just under 11 billion inner loops per second when running at $150MHz$. Simplifying the system to use a 2D coordinate system would double the number of PE's and, accordingly, the throughput. For a more complete treatment of time-delay beamforming on FPGA's, including a detailed comparison between FPGA and DSP implementations, see [2].

3 Frequency Domain Beamforming

Frequency domain techniques are commonly used to beamform selected frequencies of received signals. This makes it possible to determine frequency content *and* direction-of-arrival for a signal in the same computation. To do this, an FFT of the sensor data is first computed and then the following algorithm executed for each beam: [1]

```
formBeam(BeamDirection b) {
  for f = 0 to numFrequencies
    for (s=0;s<numSensors;s++)
      sum[b,f] += fftData[b,f] *
                    steeringWeights[b,f,s];
}
```

where *fftData* and *steeringWeights* are complex values.

A problem with the above computation is the storage of the steering weights. Consider a typical problem with $2,500$ beams, 256 frequencies, and 400 sensors. The storage required for steering weights in this case would be $1GB$.

A key observation is that the frequency domain computation outlined above is equivalent to a time-delay computation — the signals of interest are delayed prior to summing. However, in the frequency domain approach, the steering weight accomplishes this by phase rotating the complex FFT data. A way to reduce the needed steering weight memory is to store time delays as in time-delay beamforming. Steering weights (phase rotation terms) are formed on-the-fly via a multiplication of the time delay with the frequency of interest. This is shown in the following:

```
formBeam(BeamDirection b) {
  for f = 0 to numFrequencies
    for (s=0;s<numSensors;s++) {
      phaseAdjust = delay(b,s) * f;
      mag = 1.0;
      steeringWeight =
        polarToRectangular(mag, phaseAdjust);
      sum[b,f] += fftData[s,f] *
                    steeringWeight;
    }
}
```

This reduces the storage required from $1GB$ to $4MB$, making an embedded implementation possible. The price paid is the area of a multiplier and polar-to-rectangular conversion. In hardware, the conversion is readily accomplished using an unrolled/pipelined CORDIC unit [3]. The area of such a CORDIC is similar to the area of a multiplier.

The above design can be significantly improved. This is done by directly using the phaseAdjust term to rotate the

[1]For this analysis we assume the FFT is computed elsewhere as a part of the data collection process. Given the large number of beam responses typically computed from one set of FFT results, the FFT computation becomes a very small part of the overall computation.

phase of the FFT data prior to summing, and avoiding the complex multiply-accumulate. To do so the FFT data must be pre-converted to polar form. This improved computation is shown in the following:

```
formBeam(BeamDirection b) {
  for f = 0 to numFrequencies {
    for (s=0;s<numSensors;s++) {
      phaseAdjust = delay(b,s) * f;
      phase = fftPhase[s,f] + phaseAdjust;
      mag = fftMag[s,f];
      sum[b,f] +=
        polarToRectangular(mag, phase);
  }
```

The complex multiply is thus replaced by a scalar addition and the *polarToRectangular()* required previously is simply moved down in the pipeine. The net result is a circuit approximately half the size of the original. The datapath for this computation is shown in Figure 3.

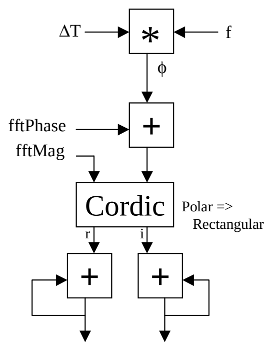


Figure 2: Frequency-Domain Beamformer Datapath

As with the time-delay derivation above, the delays are stored in off-chip memories and fed to the PE's as needed. Due to the requirement of beamforming each frequency separately, the required bandwidth to external memory is greatly reduced. That is, one delay fetch is needed only every $f = numFrequencies$ cycles. This makes it possible for the memories to support many more PE's than with the time-delay organization.

The achievable parallelism in this computation is thus limited not by memory but by the available circuit area. As with time-delay beamforming, the PE structure consists of a highly pipelined functional unit able to produce one inner loop result per clock cycle. The depth of such a pipeline, measured in clock cycles, is on the order of $n + 2$ where n is the data word width (the multiplier is $n/2$ cycles deep and the CORDIC $n/2$ cycles deep). Using Virtex-II's dedicated multipliers would further allow for a shallower pipeline as well as a smaller PE area. A concrete example of such a beamformer on Osiris would consist of 48 processors running at $150MHz$, each producing one inner loop computation per clock cycle. Using the original $1GB$ memory to hold all steering weights, each such inner loop corresponds simply to the 8 operations of a complex multiply-accumulate for a total equivalent computational throughput of 57.6 billion operations per second. However, such a design is infeasible in an embedded design due to the $1GB$ memory required. Using the polar-to-rectangular method outlined above to save steering weight storage would increase the effective ops per inner loop to 12 or more with a corresponding increase in equivalent computational throughput.

4 Matched Field Beamforming on FPGA's

The final beamforming algorithm considered is a matched field beamformer. In a shallow water environment, multipath reflections create a complex acoustic field, degrading the performance of conventional beamforming approaches. In matched field beamforming, the multipath signal replicas are used to advantage to determine not only bearing but also range and and depth to the target. This is possible because the vertical angle-of-incidence of the signal on the sensor array is a function of the water depth, the sensor depth, and the target range and depth. Given a hypothesis of the bearing, range, and depth of the signal source, the k strongest multipath signal replicas (rays) are computed and frequency domain beamforming done for each such ray and summed (in a typical problem $k = 16$). As with conventional beamforming, many such hypotheses are tested for the presence of a source. However, each such hypothesis or *beam* corresponds to a voxel of water and thus the number of beams to be formed is very large — a typical problem might require 30 million such beams.

The computation can be simplified by casting it as a two-stage computation. The first stage does matched field beamforming to relatively coarse ocean voxels. Specifically, for each voxel it determines the k strongest rays originating from that voxel and computes their relative delays at the sensor array and voxel angles-of-departure. The second stage then beamforms those k rays to each of 400 subvoxel locations within each voxel to localize the source.

Figure 3 outlines the full datapath for a 2nd stage subvoxel beamformer with a total pipeline depth of more than 50. It operates by first computing the delay a ray would experience travelling from the origin of its coarse voxel to a particular subvoxel location. This is the top third of the figure, resulting in the time delay value labelled ΔT. It repeats this for each ray, delays all the rays appropriately, and sums them together to get a final subvoxel response. This is done in the middle section of the figure which is exactly the frequency domain beamformer kernel from the previous section. Finally, the bottom portion of the figure computes the beam power and sums across time steps.

This computation was mapped to a predecessor of the Osiris board - the SLAAC1b PCI board based on Xilinx

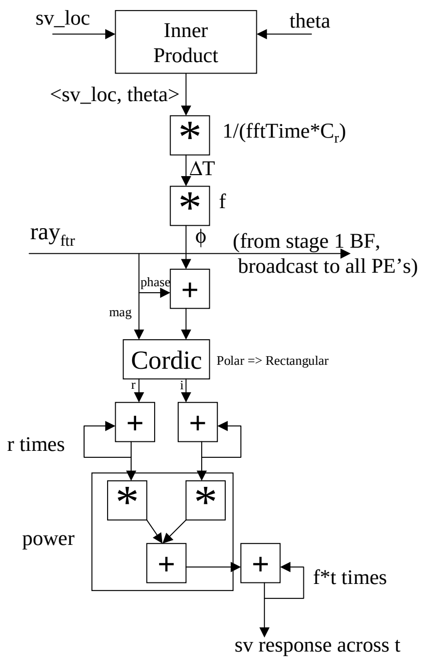


Figure 3: Matched Field Subvoxel Beamformer Datapath

4K parts (one XC4085, two XC40150's) and 10 SRAMs. The XC4085 does the first stage beamforming once a second and transmits its results to the XC40150's to perform the second stage subvoxel beamforming. A total of eight subvoxel beamformers fit into the XC40150's, operate at $50MHz$, and deliver $400M$ inner loop computations per second. A board computes $180,000$ beams — 400 subvoxels for each of 450 coarse voxels. The inner loop calculation in the subvoxel beamformer makes up the bulk of the computation and represents about 14 arithmetic operations giving a sustained equivalent computational rate of 5.6 billion arithmetic ops/second. As many boards as desired can be employed in parallel to achieve the desired area coverage. This 4K FPGA implementation was compared to that of C-code running on a variety of similar-vintage machines including Pentium-II and Pentium-III's, HP PA-RISC workstations, and a G4 Power PC. The fastest performing machine was a $552MHz$ PA-RISC workstation and its runtime was $18\times$ that of the FPGA. The slowest machine was a $400MHz$ Pentium-II machine with a runtime $83\times$ as long as the FPGA. Finally, a straightforward port of the 4K design to the Osiris board would result in a $9-12\times$ speedup ($3\times$ from clock rate and $3-4\times$ from density), bringing the sustained equivalent computational rate to as many as 67 billion arithmetic ops/second.

5 Summary

Various versions of the designs outlined above have been constructed and tested on a variety of FPGA platforms and technologies using actual array geometries and sampled data. The first was based on the frequency-domain design of Section 3 and executed on a Wildforce board (Xilinx 4K technology). It was for a linear towed array. The second was done for an air-acoustic application using a circular array and executed on a Virtex platform (it is detailed in [4]). Finally, the matched field design was completed, as stated, using the SLAAC1b platform and has been operational for 18 months. The implementation results for all three systems, after suitably adjusting for technology improvements in the Osiris platform, help to substantiate the validity of the sizing and performance estimates provided in the previous sections of this paper.

6 Conclusions and Future Work

The above results were obtained using a combination of algorithm reformulations, pipelining, and parallel processing. In all cases, the availability of local memories had a great impact on the form of the final design. Finally, all of the discussed designs were targeted to non-adaptive beamforming algorithms. The next step in this work is the investigation of processing architectures and algorithms for adaptive beamforming. Work is currently underway on developing a parameterized floating-point module library as well as the required matrix kernels for this purpose.

References

[1] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk, "The multiple wordlength paradigm," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '01)*, Kenneth L. Pocek and Jeffrey M. Arnold, Eds. IEEE Computer Society, April 2001, IEEE Computer Society Press.

[2] Paul Graham and Brent Nelson, "Fpga-based sonar processing," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, J. Cong and S. Kaptanoglu, Eds., Monterey, CA, February 1998, ACM SIGDA, pp. 201–208, ACM Press.

[3] R. Andraka, "A survey of CORDIC algorithms for fpga-based computers," in *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 1998, pp. 191–200.

[4] S. Scalera, M. Falco, and B. Nelson, "A reconfigurable computing architecture for microsensors," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '00)*, Kenneth L. Pocek and Jeffrey M. Arnold, Eds. IEEE Computer Society, April 2000, pp. 59–67, IEEE Computer Society Press.

Configurable Computing Solutions for Automatic Target Recognition

John Villasenor, Brian Schoner, Kang-Ngee Chia, Charles Zapata,
Hea Joung Kim, Chris Jones, Shane Lansing, and Bill Mangione-Smith
Electrical Engineering Department
University of California, Los Angeles
Los Angeles, CA 90095-1594

FPGAs can be used to build systems for automatic target recognition (ATR) that achieve an order of magnitude increase in performance over systems built using general purpose processors. This improvement is possible because the bit-level operations that comprise much of the ATR computational burden map extremely efficiently into FPGAs, and because the specificity of ATR target templates can be leveraged via fast reconfiguration. We describe here algorithms, design tools, and implementation strategies that are being used in a configurable computing system for ATR.

1. Introduction

The ability to rapidly modify the gate level logic of an FPGA during execution opens a number of new computing possibilities that have only recently begun to be explored. Configurable computing machines that exploit this ability will involve architectures that differ in important ways from those used in current machines, and will support a wide range of new and powerful capabilities. At the simplest level, a single FPGA can implement an arbitrary number of designs in rapid succession, and can therefore deliver the functionality of a device many times its size. More sophisticated implementations in which the configuration control receives input from the results of previous computations or from the external operating environment can also be envisioned. Finally, rapid reconfiguration makes feasible the implementation of dedicated logic circuits to support large numbers of highly specific computational tasks that are wholly unsuited to ASIC implementation.

Configurable computing requires 1) commercially available FPGAs with sufficiently fast configuration times, 2) design tools that understand and take advantage of hardware dynamisms, and 3) boards and other higher level interface and support hardware that will make fast-reconfigurable FPGAs viable in real systems. Although there is not yet a base of commercial FPGAs with submillisecond reconfiguration times to satisfy the first of these requirements, there has been prototype development in both industry and academia to explore the hardware issues of fast reconfiguration. It is our belief that this is an area which the FPGA vendors are well prepared to address, and that fast reconfiguration will receive increased commercial attention as the payoffs on the application side become clear. By contrast, design tools and systems to support use of dynamic computing devices have been in a state of relative immaturity. We describe here results from an ongoing project to build a configurable computing system for automatic target recognition (ATR). Focusing on this application has furnished quantitative results on design time and challenges, configuration overhead, and computational efficiency of configurable computing systems. More generally, it has led to an understanding of the requirements and hurdles involved in extending configurable computing to more general applications.

The rest of this paper is organized as follows: The remainder of the introduction includes a description of related work in configurable computing, and an overview of the automatic target recognition (ATR) problem that serves as the application focus for the new results presented here. Section 2 introduces the basic mapping of ATR target templates into FPGA adder trees that forms the core of the processing, and discusses some of the trade-offs in using rapid reconfiguration to support ATR. Section 3 discusses design and partitioning issues to support rapid implementation of a large set of target templates. Section 4 presents a more detailed analysis of design trade-offs and considers some the issues of I/O and board design for a FPGA-based ATR system. Conclusions and a brief description of ongoing and future work are contained in Section 5.

1.1 Overview of related work

Configurable computing has been explored in both

0-8186-7548-9/96 $05.00 © 1996 IEEE

academia and industry for several years. On the hardware side, several fast-reconfiguring FPGAs have been developed. One of these is the Configurable Logic Array (CLAy) from National Semiconductor, which is a fine-grained device consisting of an array of 56 by 56 cells, each of which is roughly equivalent to a half adder and D-flip flop. Configuration bitstreams can be loaded into the CLAy using 8 pins, allowing a complete reconfiguration of the device in approximately 750 microseconds. The CLAy also supports partial reconfiguration, which reduces the reconfiguration time linearly in accordance with the fraction of the gate array concerned.

An alternative to external loading of bitstreams used in the CLAy and in most other FPGAs is the context-switched approach advocated by a group at MIT [1,2]. In this device, referred to as a Dynamically Programmable Gate Array (DPGA), multiple configurations reside simultaneously on chip. One of these configurations occupies the active layer, and is the one which is actually executing. Any of the others can be switched to the active layer in one clock cycle. The increase in chip area to support three extra contexts is approximately 20%.

Utilization of FPGAs as dynamic computing devices has been explored by several groups including a team led by Hutchings at Brigham Young University. Hutchings has performed a series of thorough studies in which the benefits of partial reconfiguration was explored using the application example of neural nets [3]. The Brigham Young group has also investigated the use of partial reconfiguration as a means to construct a computer with a dynamic instruction set [4]. This idea, which has also been discussed by Athanas and Silverman in [5], achieves increased efficiency by using FPGA resources to hold the instructions that are needed on an application-specific basis. These experiences have led to the formulation of design methodologies for partially reconfiguring systems [6], and to important quantitative results on the benefits of partial reconfiguration. For example, for the neural net application partial reconfiguration enabled a 25% reduction in configuration time and a 50% increase in functional density compared with a system based on complete reconfiguration.

In a previous publication [7] we described the implementation of a video communications system implemented using configurable computing techniques. This system delivers real time video at a rate of 8 frames/second, and includes the steps of image transformation, quantization and run-length coding, and BPSK modulation/demodulation. These functions are implemented using a single 5000 gate CLAy, with rapid swapping of designs used to time share the gate array hardware. The rapid swapping of designs used in the video system has some commonalities, as well as some significant differences with the approach for ATR that we describe in the present paper.

1.2 Application Description: ATR

Automatic target recognition is among the most demanding real time computational problems in existence. The challenge addressed by an ATR system is conceptually simple -- to analyze a digitally represented input image or video sequence in order to automatically locate and identify all objects within the scene of interest to the observer. Since there are many types of imaging devices and many algorithmic choices available to a designer,

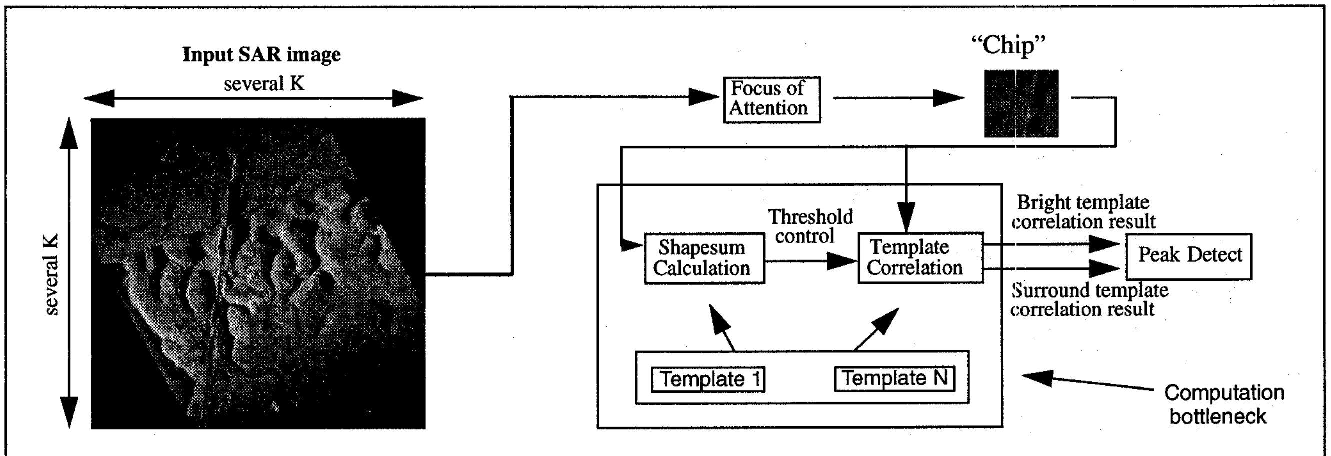


Figure 1 High level block diagram for ATR processing. The focus of attention algorithm identifies regions of interest, referred to as "chips", in SAR images. Chips are correlated against a series of binary target template pairs, with each pair containing a bright template (identifying pixels of strong expected radar return) and a surround template (strong radar absorption). Templates with highest correlation are selected in the peak detection step.

there are clearly a large number of possible ways to implement an ATR system. In this paper we focus on a particular approach which is currently being applied in the U.S. Department of Defense Joint STARS airborne radar imaging platform, and which therefore has high current relevance and interest.

The processing used in ATR is illustrated in simplified format in Figure 1. Synthetic aperture radar (SAR) images consisting of 8-bit pixels and measuring several thousand pixels on a side and are generated in real time by the radar imager. Images are input to a focus-of-attention processor which identifies a set of regions of interest, each of which contains a potential target. These regions of interest, known by the potentially confusing term "chip", must then be correlated with a very large number of target templates. Target templates are binary; e.g. each pixel is represented using one bit. The correlation results are output to a peak detector which identifies the template and relative offset at which the peak correlation value occurs. The correlation of chips with templates is the computational bottleneck in the system, involving data rates and computational requirements that exceed by several orders of magnitude the processing load in any other steps in the algorithm. While the precise system parameters vary with implementation, in the work described here we use chip sizes of 128 by 128 and template sizes of 16 by 16. A correlation of a single chip with a single template in this case involves consideration of approximately 128^2 relative offsets, corresponding to 10^5 bits of output data if the correlation outputs are represented using 6 bits. If there are 10^3 templates to be evaluated per chip, the magnitude of the processing task becomes readily apparent when one considers that the imaging system produces many frames per second, each of which contains many chips.

Figure 2 illustrates the correlation operation targeted for FPGA implementation in more detail. Target templates occur in pairs, one member of which is called the bright template and contains pixels from which a strong radar return is expected, and the other member of which is the surround template and identifies pixels where strong radar absorption is expected. In both cases the template is of size 16 by 16, with pixels represented using only one bit. The templates tend to be sparsely populated, with only a relatively small percentage of the pixels set to 1. As will be discussed later, this property is important in obtaining high performance in FPGA implementations. The first step of the correlation is known as a shapesum calculation, in which the 8-bit SAR chip is correlated with the bright template, providing for every pixel in the chip a number which is used for local gain control. The second step is the actual correlation, which is performed in parallel on eight different binary images, each of which is created by applying a different threshold to the chip. The binary images are correlated with both the bright template and the surround template, producing eight pairs of correlation outputs. The shapesum value is used to select which output pair will be processed in the peak detection step.

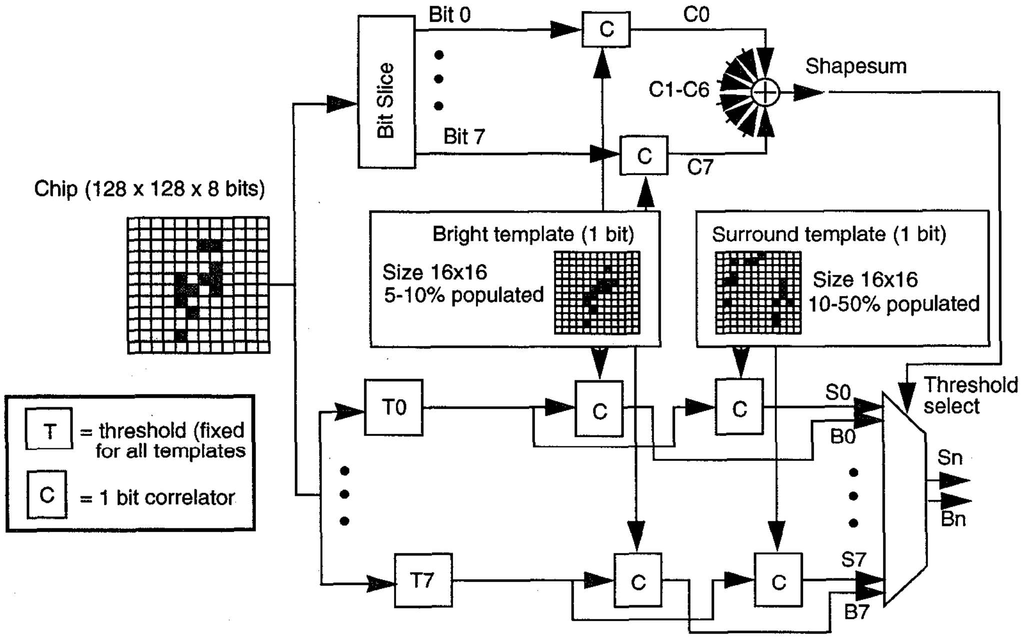


Figure 2 Correlation operations showing shapesum calculation (top) and parallel correlations (bottom)

2. Mapping and dynamic reconfiguration of target templates

FPGAs offer an extremely attractive solution to the correlation problem. First of all, the operations being performed occur directly at the bit level and are dominated by shifts and adds, making them easy to map into the hardware provided by the FPGA. This contrasts, for example, with multiply-intensive algorithms which would make relatively poor utilization of FPGA resources. More importantly, the sparse nature of the templates can be utilized to achieve a far more efficient implementation in the FPGA than could be realized in a general purpose correlator. This can be illustrated using the example of the simple template shown in Figure 3.

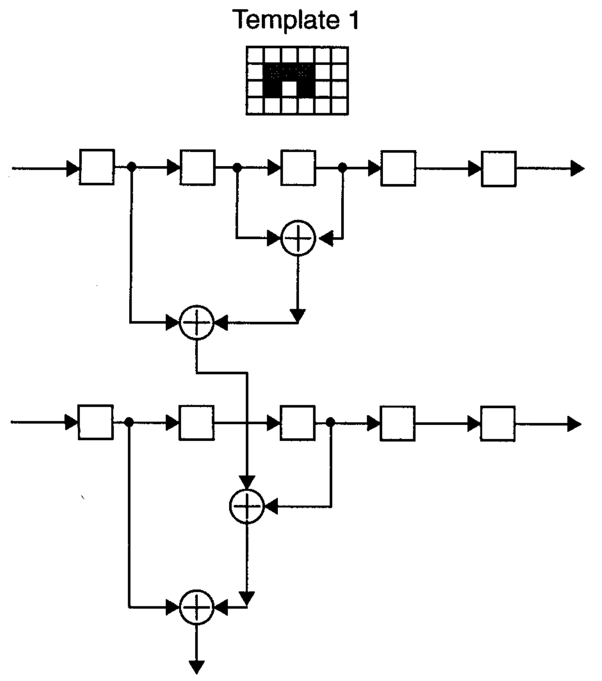


Figure 3 Example binary template with five "on" pixels (top) and corresponding adder tree (bottom).

In this example template, only 5 of the 24 pixels are "on". At any given relative offset between the template and chip, the correlation output is the sum of the five binary pixels in the chip that lie immediately above the "on" pixels in the template. The template can therefore be implemented in the FPGA as a simple adder tree as shown in Figure 3. The chip pixel values can be stored in flip-flops, and are shifted to the right by one flip flop with each clock cycle. Though correlation of a large image with a small mask is often understood conceptually in terms of the mask being scanned across the image, in this case the opposite is occurring - the template is hard-wired into the FPGA while the image pixels are clocked past it.

Another important opportunity for increased efficiency lies in the potential to combine multiple templates on a single FPGA. The simplest way to do this is to spatially partition the FPGA into several smaller blocks, each of which handles the logic for a single template. Alternatively, one can seek to identify templates having some topological commonality, and which can therefore share parts of adder trees. This is illustrated in Figure 4, which shows two templates which share several pixels in common, and which can be mapped using a set of adder trees which leverage this overlap.

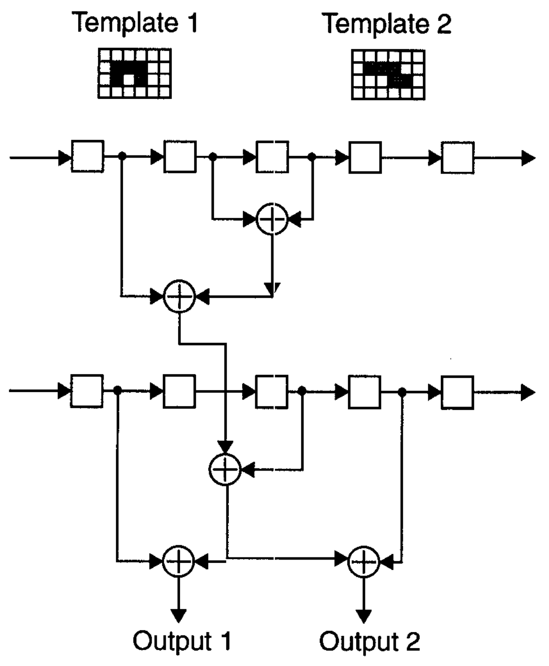


Figure 4 Template commonalities are exploited to reduce hardware requirements for computing multiple correlations.

The advantage of using FPGAs over ASICs is that FPGAs can be dynamically optimized at the gate level to exploit template characteristics. An ASIC would have to provide large general purpose adder trees to handle the worst case condition of summing all possible template bits. The FPGA, however, exploits the sparse nature of the templates, and only constructs the small adder trees required. We have also shown that FPGAs can exploit other factors such as collapsing adder trees with common elements, and packing unused data points into space-saving RAM-based shift registers. The end result is that a single FPGA can efficiently compute several templates in parallel more efficiently than several general purpose correlating ASICs.

There are many factors that determine the performance gain that results from using an FPGA. One of the most important is FPGA reconfiguration time. Assuming that performing the correlation of a single chip requires approximately $128^2 = 16K$ clock cycles, the reconfiguration must be performed in $\sim 10^3$ or fewer clock cycles to avoid prohibitive overhead. In this respect a context-switched FPGA with two contexts would be extremely useful. If the idle context could be loaded while the active context was processing, then reconfiguration overhead would vanish. The achievable parallelism is also a critical parameter. Based on the work to date, we estimate that we can map an average of 20 bright templates or 5 surround templates on a single 13000-gate FPGA.

2.1 Experimental results for FPGA resource utilization

The approach of using a template-specific adder tree achieves significant reduction in routing complexity over a general correlator which must include logic to support arbitrary templates. To a first approximation, the extent of this reduction is inversely proportional to the fraction of "on" pixels in the template. While this complexity reduction is important, it alone is not sufficient to lead to efficient implementations on FPGAs. This is due primarily to the limited number of flip flops available on commercial FPGAs (for example, the Xilinx XC4010 and ATT ORCA 2C10 contain 800 and 1024 flip flops respectively). This would not generally be sufficient to support buffering the 112 pixels per chip row that are not actually under the template, but need to be wrapped around to the next row of the template. The total number of 1-bit storage elements needed to hold buffered pixel values for all 16 rows is 16 * 112 = 1792. Implementing these on the FPGA using the usual flip-flops based shift registers is inefficient, and for many FPGAs impossible.

This problem can be resolved by collapsing the long strings of image pixels that are not being actively correlated against a template into shift registers, which can be implemented very efficiently on some look-up-table based FPGAs. For example, RAMs in the Xilinx XC4000 library can be used as shift registers which delay data by some predetermined number of clock cycles. Each 16x1 bit RAM primitive uses up a function generator on the FPGA, and can implement an element which is effectively a 16-bit shift register in which the internal bits cannot be accessed. A flip-flop is also needed at the output of each RAM to act as a buffer and synchronizer. A single control circuit is used to control the stepping of the address lines and the timely assertion of the write-enable and output-enable signals for all the RAM-based shift register elements. This is a small over head price to pay for the savings in CLB usage relative to a brute force implementation using flip flops.

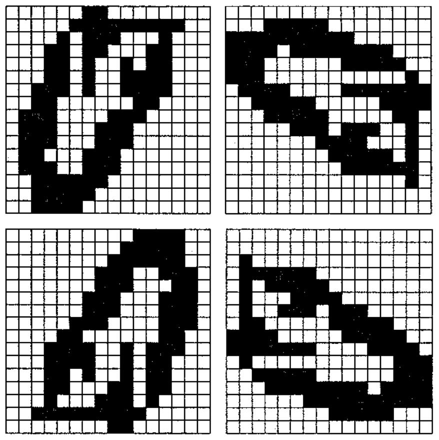


Figure 5 Four templates that were mapped onto the Xilinx 4010 to explore resource utilization. These examples were generated by 4 rotations of a synthetic template. The number of "on" pixels is 91, which is higher than what would be expected for most templates.

By contrast, the 256 image pixels that lie within the 16 by 16 template boundary at any given time can be stored easily using flip-flop based registers, since there are sufficient flip-flops available to do this, and the adder tree structures do not consume flip-flops. Also, using standard flip-flop based shift registers for image pixels within the template simplifies the mapping process by allowing access to every pixel in the template. New templates can be implemented by simply connecting the template pixels of concern to the inputs of the adder tree structures. This leads to significant simplification of automated template mapping tools.

To gain a fuller understanding of the FPGA resource trade-offs involved in template mapping, we implemented in parallel the four templates shown in Figure 5 onto the Xilinx 4010 using the techniques described above. Each template had 91 "on" pixels, few of which are shared with other templates. Since parallelism was low, this exercise gaves a worst-case estimate for the capacity of the 4010. The resulting FPGA resource utilization, as summarized in Table 1, shows that 318 flip-flops and 756 function generators were used. The resources used by the two components of target correlation, namely storage of active pixels on the FPGA (1st row of table) and implementation of the adder tree corresponding to the templates (2nd row) are independent of each other. The resources used by the pixel storage are determined by the template size, and are independent of the number of templates being implemented. Adding templates involves adding new adder tree structures, and will hence increase the number of function generators being used. The total number of templates implementable

	Flip Flops	Fcn. Gen.	I/O pins
Pixel storage	318	116	4
Adder trees	0	640	28
Total used	318	756	32
Total available	800	800	160

Table 1 Xilinx 4010 resource utilization for the four sample templates shown in Figure 5. Resources for the two basic functions implemented (pixel storage and adder trees) are shown separately. The adder trees account for the majority of the function generator utilzation, and are a key factor in limiting the number of templates that can be implemented simultaneeously in a single configuration.

on an FPGA will be bounded by the number of usable function generators.

The four templates in the example above exhibit a very low number of common pixels and would probably not be grouped together by the partitioning algorithm. These results suggest that in practice, we can expect to fit 6-10 surround templates having a higher number of overlapping pixels onto a 13000 gate FPGA. Since the bright templates are less populated than the surround templates, we estimate that 15-20 bright templates can be mapped onto a 13000 gate FPGA.

3. ATR template partitioning issues

To minimize the number of FPGA reconfigurations necessary to correlate a given target image against the entire set of templates, it is necessary to maximize the number of templates placed in every configuration of the FPGA. To accomplish this optimization goal, we want to partition the set of templates into groups that can share computations (i.e., share adder trees) so that fewer resources are used per template. Since the set of templates may number in the thousands, and the goal may be to place ten to twenty templates per configuration, exhaustive enumeration of all of the possible groupings is not an option. Instead, it is best to seek a heuristic method which will furnish a good, although perhaps suboptimal, solution.

Correlation between two templates can establish the number of pixels in common, and is a good starting point for comparing and selecting templates. However, some extra analysis beyond performing iterative correlations on the template set is necessary. For example, a template with a large surface area may correlate well with several smaller templates, perhaps even completely subsuming them, but the smaller templates may not correlate with each other at all, and would involve no redundant computations. There are two possible solutions to this. The first is to ensure that any template added to an existing group is approximately the same size as templates in the group. The second option is to compute the number of additions required each time a new template is brought in - effectively recomputing the adder tree every time.

Recomputing the entire adder tree is computationally expensive, and is not a good method of partitioning a set of templates into subsets. However, one of the heurstics used in deciding whether or not to include a template into a newly formed partition is to determine the number of new terms that adding the template would create in the partition's adder tree. The assumption is that more terms would result in a significant number of new additions, resulting in a wider and deeper adder tree. By keeping the number of new terms created to a minimum, newly added templates do not increase the number of additions by a siginificant amount. Using C++, we have created a design tool to implement the partitioning process. C++ allows us to abstract templates and sets of templates into easily manipulated objects. New methods of comparison can be implemented by adding new methods to the objects, making this system adaptable to different heuristics. Furthermore, the system is not tied down to any particular platform, as C++ compilers are available for many different platforms.

The tool uses an iterative approach to partitioning templates. Templates which compare well to a chosen "base" template (usually selected by largest area) are removed from the main template set and placed in a separate partition. This process is repeated until all templates are partitioned. After the partitions have been selected, the design tool computes the adder tree for each partition. Figures 6a-6c show the process of creating an adder tree from the templates in a partition. Within each partition, the templates (shown in Figure 6a) are searched for shared subsets of pixels (Figure 6b). These subsets, called terms, can be automatically added together, leading to a template description using terms instead of pixels (Figure 6c). The most commonly occurring addition of two terms is chosen to be added together, forming a new term which can be used by the templates. In this way, each template is rebuilt by combining terms in such a way that the most redundant additions are shared between templates, and the final result of the process are terms which compute entire templates. For the sample templates shown here, 54 additions would be required to compute the correlations for all five templates in a naive approach. However, after combining the

Figure 6: Example of template grouping to leverage pixels common to multiple templates. In this example, the grouping reduces the number of adders needed from 54 to 29.

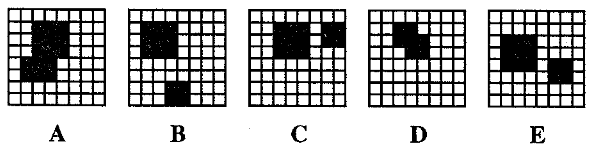


Figure 6a: Five sample templates to be allocated on the same FPGA configuration.

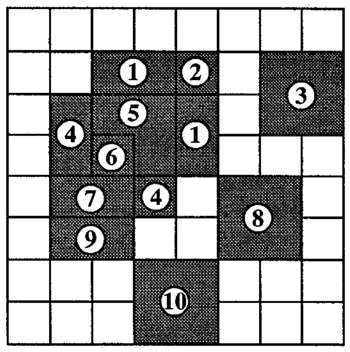


Figure 6b. The terms that result from overlapping templates. Each term corresponds to a group of pixels needed by a particular set of templates. These terms correspond to nodes on an adder tree.

Template A	① + ② + ⑤ + ⑥ + ⑦ + ⑨
Template B	④ + ⑤ + ⑥ + ⑦ + ⑩
Template C	① + ② + ③ + ⑤ + ⑥
Template D	① + ⑤
Template E	④ + ⑤ + ⑥ + ⑦ + ⑧

Figure 6c. The templates rewritten as sums of terms. The most common additions are coalesced into new terms. Here, 5 + 6 occurs four times, and is the first candidate to be replaced with a new term, 11.

templates through the process described here, only 29 additions are required.

4. Configurable computing system design

The increased performance of configurable systems comes with several costs. These include the time and bandwidth required for reconfiguration, memory and I/O required to store intermediate results, and additional hardware required for efficient implementation and debugging. Minimizing these costs requires innovative approaches to system design.

Figure 7 compares the configurable computing architecture discussed here (7c) with a traditional processor (7a) and a customizable computing architecture (7b). The traditional processor receives simple operands from a data memory, performs a simple operation in the program, and returns the result to data memory. Custom computers attempt to gain performance advantages over traditional processors by implementing common operations in hardware. Some fine-grain systems have been proposed [4] but the majority of successful custom computers implement complex operations in a large array of FPGAs [8,9] as depicted in Figure 7b. Large custom computers deliver tremendous performance for some applications, but have definite shortcomings. The number of FPGAs is typically fixed, or only slightly variable through the use of parallel cards. Large applications may not fit, and small applications may inefficiently use available resources. Also, large

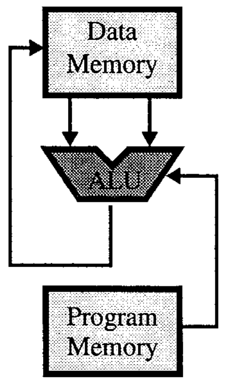


Figure 7a
Traditional Processor

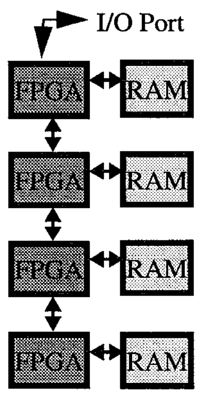


Figure 7b
Custom Computer

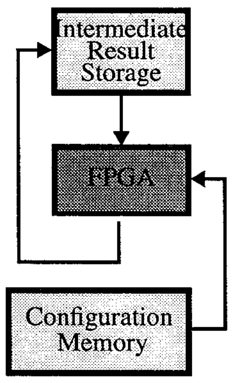


Figure 7c
Dynamic Computer

Figure 7 Architecture comparisons. Traditional processors (7a) take simple operands from memory, perform an operation, and return the result to memory. Custom computers (7b) gain performance advantages by putting common operations into hardware. In dynamic machines (7c) logic is programmed by selecting configuration bitstreams stored in nearby RAM.

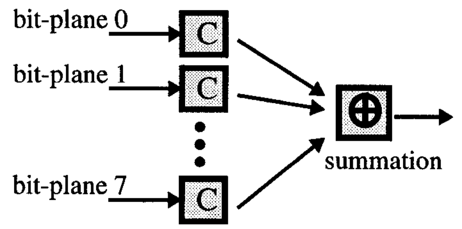


Figure 8a Eight FPGAs, each performing one bit correlations

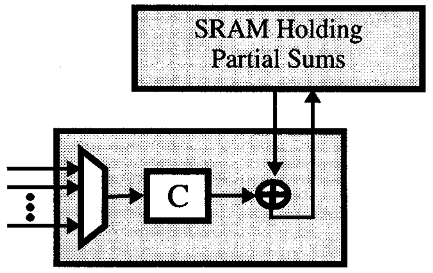


Figure 8b One FPGA, performing the correlations and adding the results serially

custom computers can be difficult to partition and program.

Our approach to configurable computing is to use a small number of rapidly reconfiguring FPGAs tightly coupled to an intermediate result memory and a configuration memory. This configurable computing architecture attempts to gain some of the advantages of a traditional programmable processor and some of the application-specific performance of a custom computer. We have had success implementing a video compression and transmission system to this architecture [7], and will now briefly describe some of the trade-offs involved in implementing an automatic target recognition (ATR) algorithm on this architecture.

Partitioning of computations plays a major role in any parallel processing environment. For a configurable computing system, partitioning is particularly interesting because it is both a hardware and a software issue. Consider two methods for performing addition of shapesum bit-planes shown in Figure 8a. Method A is a straightforward parallel implementation requiring several FPGAs, and has several drawbacks. First, the outputs from several FPGAs converge at the addition operation. This may create a severe I/O bottleneck. Second, the system is not scalable. If the system requires more precision, and therefore more bit-planes, more FPGAs must be added to the system.

Method B in Figure 8 illustrates our approach. Each bit plane is correlated individually, and then added to the previous results in the temporary storage. Method B is completely scalable to any image or template precision, and this simple architecture can implement all the correlation, normalization, and peak detection routines required for ATR. One drawback of method B is the cost and power required for the wide temporary result SRAM. Another possible drawback is the extra execution time required to run ATR correlations in serial. The ratio of performance to number of FPGAs is roughly equivalent for the two methods, and the performance gap can be closed by simply using more of the smaller method B boards.

The approach of a reconfigurable FPGA connected to an intermediate memory allows us to have a fairly complicated flow of control. For example, the shapesum calculation in ATR tends to be more difficult than the image-template correlation. A single FPGA might compute two shapesums or four correlations. For this reason, one may wish to have a program that performs two shapesum operations and forwards the results to a single correlation operation as shown in Figure 9. Looping and branching operations can also find uses for adjustable precision and adaptive filtering.

Reconfigurations for 10k gate FPGAs are typically around 20kB in length. Reconfiguring every 20ms gives a reconfiguration bandwidth of approximately 1MB per FPGA per second. This reconfiguration bandwidth, coupled with the complexity of the flow of control can be handled by placement of a small microcontroller and configuration RAM next to every FPGA. The microcontroller permits complicated flow of control, and since the microcontroller addresses the configuration RAM, it frees up valuable I/O on the FPGA. The microcontroller also is important for debugging, which is a major issue in config-

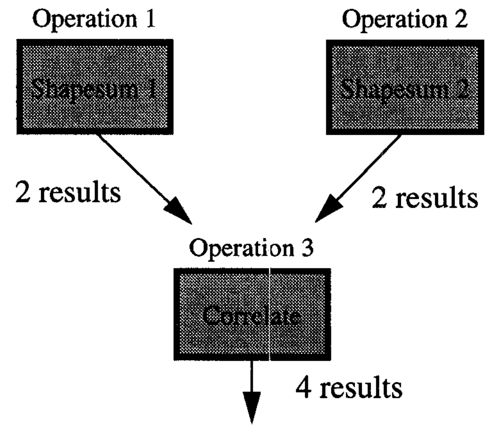


Figure 9 Reconfiguration Flow of Control Example

urable systems because the many different hardware configurations can make, it difficult to isolate problems.

Figure 10 shows the current (as of April 1996) version of the evolving configurable computing system for performing ATR. The principal components are labeled in the graph and include a "dynamic" FPGA which is reconfigured on the fly, and which performs most of the computing functions, and a "static" FPGA which is configured only once, and which performs control and some computational functions. The EPROM holds configuration bitstreams, and the SRAM holds the input image data (e.g. the chip). Because the correlation operation involves the application of a small target template to a large chip, a FIFO is needed to hold the pixels that are being wrapped around to the next row of the template mask. The template sizes used in this implementation are of size 8 by 8; the image against which the templates are correlated are 128 by 128. The maximum clock rate in the current design is 12.5 MHz. Each configuration of the dynamic FPGA implements a total of four template pairs (four bright templates and four surround templates).

The shapesum (see Figure 2) computation is performed in parallel using the approach illustrated in Figure 4. This requires a total of D clock cycles, where D is the depth of representation of each pixel. Once the shapesum results are obtained, the correlation outputs are produced at the rate of 1 per clock cycle. Parallelism can not be as directly exploited in this step because different pixels are "on" for different templates. However, in the limit of very large FPGAs the number of clock cycles to compute the correlation is upper bounded by the number of possible thresholds as opposed to the number of templates.

5 . Conclusions

The combination of efficient implementation of bit-level processing tasks and rapid reconfiguration makes FPGAs ideally suited to problems such as ATR. By associating each target template with a relatively simple adder tree and merging topologically similar templates, a processing parallelism of approximately 10 can be achieved using currently available FPGAs. Provided that the FPGA reconfiguration time is on the order of milliseconds, dynamic reconfiguration can be used to support sustained computation for ATR template matching at much higher throuput and lower cost than a general purpose correlator.

A demonstration system for implementing the ATR algorithm illustrated in Figure 2 has been constructed and is shown in Figure 10. This system includes a single dynamically-reconfigured FPGA (Xilinx 4013) for computation and a second FPGA for control and some post-processing functions. Each configuration of the dynamic FPGA performs the ATR matching of a 128 by 128 image against four template pairs, with each template of size 8 by 8. These results provide a proof-of-concept for the via-

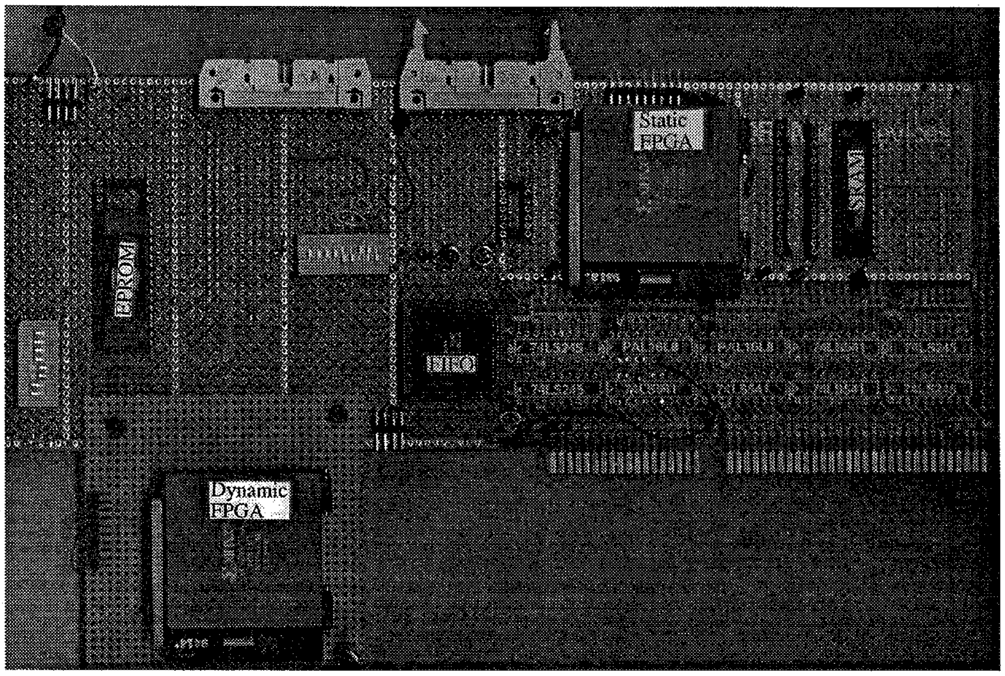


Figure 10 Configurable computing system for ATR. Most computational functions are performed on the dynamic FPGA. The static FPGA controls the processing and also performs some of the postprocessing. Other components on the board include an EPROM for configuration bitstreams, a FIFO for storage of "wraparound" pixels during the 2D correlation operation, and an SRAM for image storage. The board currently runs at a clock speed of 12.5 MHz, and correlates one 8-bit deep 128 by 128 image against sixteen 8 x 8 template pairs in approximately 210 msec (about 130 msec is used for configuration).

bility of performaing ATR in a configurable computing environment.

6. Acknowledgment

This work was supported by ARPA/ITO under contract number DABT63-95-C-0102.

7. References

[1] E. Tau, I. Eslick, D. Chen, J. Brown and A. DeHon, "A first generation DPGA implementation," *Proceeding of the Third Canadian Workshop on Field-Programmable Devices*, pp. 138-143, May 1995.

[2] A. DeHon, "DPGA Utilization and Application," to appear in *Proceedings of the 1996 International Symposium on Field Programmable Gate Arrays*, February 1996.

[3] M.J. Wirthlin and B.L. Hutchings, "Density enhancement of a neural network using FPGAs and run-time reconfiguration," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180-188, April 1994.

[4] M.J. Wirthlin and B.L. Hutchings, "A dynamic instruction set computer," *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 99-107, April 1995.

[5] P.M. Athanas and H.F. Silverman, "Processor reconfiguration through instruction set metamorphosis," *Computer*, vol. 26, pp. 11-18, March 1993.

[6] J.D. Hadley and B.L. Hutchings, "Designing a partially reconfigured system," *Proceedings of SPIE Photonics East: FPGAs for Fast Board Development and Reconfigurable Computing*, October 1995.

[7] J.D. Villasenor, B. Schoner, and C. Jones, "Video communications using rapidly reconfigurable hardware," *IEEE Transactions on Circuits and Systems for Video Technology*, January 1995.

[8] R. Amerson, R.J. Carter, W.B. Culbertson, P. Kuekes, and G. Snider, "Teramac - Configurable custom computing," *Proceedings of the 1995 Symposium on FPGAs for Custom Computing Machines*, pp. 180-188, April 1995.

[9] L. Moll, J. Vuillemin, and P. Boucard, "High-energy physics on DECPeRLE-1 programmable active memory," *ACM Third International Symposium on Field-Programmable Gate Arrays*, pp. 47-52, Monterey, CA 1995.

Design Issues for Hardware Implementation of an Algorithm for Segmenting Hyperspectral Imagery

James Theiler[1], Miriam Leeser[2], Michael Estlick[2], and John J. Szymanski[1]

[1]Space and Remote Sensing Sciences Group
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

[2]Department of Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts 02115

ABSTRACT

Modern hyperspectral imagers can produce data cubes with hundreds of spectral channels and millions of pixels. One way to cope with this massive volume is to organize the data so that pixels with similar spectral content are clustered together in the same category. This provides both a compression of the data and a segmentation of the image that can be useful for other image processing tasks downstream.

The classic approach for segmentation of multidimensional data is the k-means algorithm; this is an iterative method that produces successively better segmentations. It is a simple algorithm, but the computational expense can be considerable, particularly for clustering large hyperspectral images into many categories. The ASAPP (Accelerating Segmentation And Pixel Purity) project aims to relieve this processing bottleneck by putting the k-means algorithm into field-programmable gate array (FPGA) hardware.

The standard software implementation of k-means uses floating-point arithmetic and Euclidean distances. By fixing the precision of the computation and by employing alternative distance metrics (we consider the "Manhattan" and the "Max" metrics as well as a linear combination of the two), we can fit more distance-computation nodes on the chip, obtain a higher degree of fine-grain parallelism, and therefore faster performance, but at the price of slightly less optimal clusters. We investigate the effects of different distance metrics from both a theoretical (using random simulated data) and an empirical viewpoint (using 224-channel AVIRIS images and 10-channel multispectral images that are derived from the AVIRIS data to simulate MTI data).

Keywords: hyperspectral, k-means, image segmentation, image processing, field-programmable gate array (FPGA)

1. INTRODUCTION

Hyperspectral images, with detailed spatial resolution and upwards of 200 spectral channels, are becoming an increasingly common data product in remote sensing applications. These data provide important and useful information, but the challenge for the data analyst is to identify the important and useful features without being overwhelmed by the sheer volume of the data. One approach is provided by algorithms which segment the image by clustering pixels into classes, based on the spectral similarity of each pixel to other members of the class. This can represent a massive, albeit lossy, compression of data. For instance, we have been working with AVIRIS data,[1] in which each 614x512 pixel image has 224 16-bit channels; about 140 MB of data. When clustered into 16 classes, each pixel is represented by a 4-bit pointer to its class. The resulting image is about 150KB – a compression factor of three orders of magnitude.

As well as reducing the data for quicklook views, clustering also provides an organization of the data that can be useful for other processing downstream. Several authors have shown that clustering the data beforehand increases the performance of algorithms which attempt to "learn" features from a small number of examples.[2,3] Schowengerdt[4] suggests the use of image segmentation for change detection: a change in the segmentation is more likely to indicate an actual change on the ground, since the segmentation is relatively robust to changes in sensor performance and atmospheric conditions. It has also been argued that signal-to-clutter ratios can be improved by treating individual clusters separately; the variance within a cluster is generally much smaller than the variance over the whole image.

While there are clear benefits to clustering high-dimensional data sets, workstation implementations of clustering algorithms can be slow. Most algorithms, such as k-means, are iterative and require numerous passes through the data before convergence is achieved. Each iteration requires a computation of distance from every data point to every cluster center, and each distance requires accessing all the spectral channels. To accelerate this computation, we are developing an implementation of the k-means algorithm on reconfigurable hardware, using FPGA (Field Programmable Gate Array) chips. Reconfigurable hardware provides the ability to exploit the fine-grained parallelism inherent in the computation, while maintaining the flexibility to consider variants on the design and on the algorithm. FPGAs are particularly well suited to this application because the amount of parallelism and processing element bit widths can adapt to the task, allowing the designer to take maximum advantage of the hardware at hand. In addition, clustering on an FPGA board frees the workstation for the analyst to apply other algorithms to the data.

Implementations in hardware involve a different set of design tradeoffs than implementations in software. For example, the main goal of a software implementation may be to minimize the number of iterations. In hardware, our goals are to simplify the underlying operations as much as possible in order to speed up the calculations and to be able to provide more parallelism. A simpler operation translates to a smaller area datapath which in turn translates to more copies replicated on the chip. To this end, we are investigating alternative distance measures. By using a Manhattan or a Max distance, instead of the more expensive Euclidean distance, we trade theoretical optimality for practical efficiency. The Manhattan and Max distance calculations both eliminate a multiplier, thus saving area. They also reduce the number of bits needed to store intermediate calculations, minimizing the area required for routing.

This paper describes a set of experiments to compare Euclidean and alternative metrics for assigning data to a cluster. We conduct two sets of experiments. The first is a set of data-independent experiments that show an idealized setup for estimating misclassification rates and their effect on the total within-class variance. The second set of experiments shows how the use of alternative distance metrics affects total variance for some AVIRIS[1] and simulated MTI[5] data sets.

2. K-MEANS CLUSTERING

Given a set of N pixels, each composed of D spectral channels, and represented as a point in D-dimensional Euclidean space (that is, $\mathbf{x}_n \in \mathcal{R}^D$, with $n = 1, \ldots, N$); we partition the pixels into K clusters with the property that pixels in the same cluster are spectrally similar. Each cluster is associated with a "prototype" or "center" value which is representative of (and close to) the pixels in that class.

One measure of the quality of a partition is the within-class variance; this is the sum of squared (Euclidean) distances from each pixel to that pixel's cluster center.

For a fixed partition, the optimal (in this sense of minimum within-class variance) location for each center is the mean of all pixels in each class. And for a fixed choice of centers, the optimal partition assigns each point to the cluster whose center is closest. The k-means clustering algorithms (there are several variants) provide an iterative scheme that operates over a fixed number (K) of clusters, while attempting to simultaneously optimize center locations and pixels assignments.

From an initial sampling, the algorithm loops over all the data points, and reassigns each to the cluster whose center it is closest to. After a full pass through the data, the cluster centers are recomputed. Note that other variants of k-means update the cluster centers each time a point is reassigned to a new cluster. This leads to faster convergence, but is more difficult to implement in hardware. Each iteration reduces the total within-class variance for the clustering, so it is guaranteed that after enough iterations, the algorithm will converge, and further passes will not reassign points. It bears remarking that this is a local minimum; and it bears further remarking that the final convergence can depend somewhat sensitively on the initialization of the algorithm.

3. ALTERNATIVE DISTANCE MEASURES

Points are assigned to the cluster centers to which they are closest; for the minimum-variance criterion, "closest" is defined in terms of the Euclidean distance. Consider a point $\mathbf{x}$ and cluster center $\mathbf{c}$ where i indexes the spectral components of each. The Euclidean distance is defined as:

$$\|\mathbf{x}-\mathbf{c}\|^2 = \sum_i |x_i - c_i|^2. \tag{1}$$

But other distance measures can also be used; for instance, the general family of p-metrics (for which the Euclidean distance is the special case $p = 2$) is given by:

$$\|\mathbf{x}-\mathbf{c}\|^p = \sum_i |x_i - c_i|^p. \tag{2}$$

To perform a k-means iteration, one must compute the distance from every point to every center. If there are N points, K centers, and D spectral channels, then there will be $O(NKD)$ operations. For the Euclidean distance, each operation requires computing the square of a number.

The Euclidean distance has several advantages. For one, the distance is rotationally invariant. Furthermore, minimizing the Euclidean distance minimizes the within-class variance. On the other hand, the Euclidean distance is more expensive than the alternatives that we are considering. The Manhattan distance, corresponding to $p = 1$, is the sum of absolute values of the coordinate differences; the Max distance, corresponding to $p = \infty$ is the maximum of the absolute values of the coordinate differences.

Neither of the alternative distances requires any multiplication, and the Max distance has the slight advantage that the number of bits required to express the total distance does not increase with the dimension D (for the Manhattan distance, we require $\log_2 D$ extra bits for the total distance over and above the bits required for the coordinate differences). Note that the Euclidean distance requires D extra bits (unless a square root is invoked!); so it is not only more computationally demanding, it also requires a wider data path than the alternative distance metrics.

In assigning a data point to a cluster center, we assign to the center which is closest according to one of the computationally cheaper metrics. We expect that in general closeness in the Manhattan or Max metric implies closeness in the Euclidean metric, but we recognize that this is an approximation that may lead to non-optimal clusterings. See Figure 1.

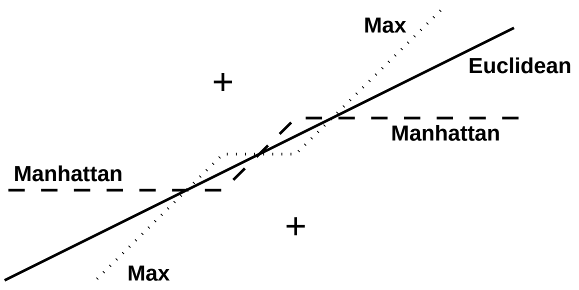


Figure 1. Two cluster centers are indicated by the two crosses in the figure above. The solid line partitions the area into points closer to the top cross and those closer to the bottom cross, where closeness is defined in terms of the Euclidean distance metric. If distance is defined in terms of a Manhattan metric, then the partition is given by the dashed line; if distance is given by the Max metric, then the partition is given by the dotted line. Points in the figure that are above the solid line, but below the dashed line will be misclassified by the Manhattan metric as belonging to the bottom class, when in fact they would be assigned by the Euclidean metric to the top class.

Our third alternative takes advantage of the fact that these two distances have values of p on either side of the $p = 2$ that corresponds to the Euclidean distance. By taking a linear combination of the Manhattan and the

Max distances, we can approximate the Euclidean distance more accurately than either of the two distances can individually. Further, the linear combination is straightforward to implement in hardware, and does not require any extra bit-width. The linear combination is given by

$$\|\mathbf{x} - \mathbf{c}\| = \alpha \max |x_i - c_i| + (1 - \alpha) \sum_{i=1}^{D} |x_i - c_i|. \tag{3}$$

The use of this linear combination was inspired by a pair of papers by Filip,[6,7] looking at linear and piecewise linear estimates of $\sqrt{x^2 + y^2}$ as a function of $x \leq y$. But where Filip's attention was restricted to two dimensions, our linear combination applies in arbitrary dimension; also, while Filip was interested in actual approximations, we are happy to approximate any monotonic function of the Euclidean distance, since our interest is in identifying which of two (or more) points is nearest to a given point.

4. DATA-INDEPENDENT ASSESSMENT OF ALTERNATIVE DISTANCE METRICS

In assessing this tradeoff, we performed an experiment to estimate how often points would be mis-assigned because a cheaper distance metric was used. The effect of metric choice on the quality of a clustering depends on many factors: the number of clusters, the dimension of the space, and the nature of the data in that space. To eliminate as many of these factors as possible, we considered an idealized situation in which three points are placed at random on the surface of a D-dimensional sphere (so that there are no "edge effects"). Two of the points are taken to be cluster centers, and the third is a data point. We compute the Euclidean distances from the data point to the two centers in order to determine which is truly closer. Then we compute the distances from the data point to the two centers using an alternative metric (Manhattan, Max, or linear combination thereof), and note whether this second pair of distances correctly identified the closest center. From a set of 10^5 such trials, we compute two statistics: relative variance and misclassification rate. The relative variance is the ratio of within-class variance for the cluster assignments provided by the alternative metric, divided by the within-class variance that would have been obtained if Euclidean distances had been used to assign points to centers. That the value is always larger than one reflects the fact that the Euclidean distance is the optimal choice for minimizing within-class variance. We estimated the misclassification rate by counting the fraction of trials for which the cheaper metric assigned the point incorrectly.

The results are shown in Fig. 2 as a function of the linear parameter α. Here $\alpha = 0$ corresponds to a pure Max metric, and $\alpha = 1$ corresponds to the Manhattan metric. The linear combination provides distances that are better than both the Max and Manhattan distances, and based on these results, we adopt $\alpha = 0.25$ as our "standard" value; it is near optimal over a wide range of dimensions D, and has the further advantage, for hardware implementation, that its denominator is a small power of two.

In Fig. 3, we plot the relative variance and misclassification rate as a function of the number D of spectral channels. As the dimension D increases, the mis-classification rate also increases. For the Manhattan metric (and for the $\alpha = 0.25$ linear-combination metric), this rate saturates at about fifteen percent, but for the Max metric, the error rate begins to approach fifty percent. That is, for very large dimension D, the Max metric is not much better than just assigning points to clusters at random.

We also performed this experiment for $K = 16$ classes. In this case, a trial consisted of 17 points placed at random on a D-dimensional sphere. Sixteen of the points were considered cluster centers, and the seventeenth point was the pixel whose class was assigned according to which of the 16 centers it was closest. The results of a Monte-Carlo experiment with 10^5 trials are shown in Fig. 4. The results are qualitatively similar to the $K = 2$ case, with the relative variance and the misclassification rate both larger in the large K case. Again, as D increases, the Max metric appears to be approaching a misclassification rate of 15/16 (that is, random assignment of centers to points), but the Manhattan and linear combination both saturate at around 40%; this is a pretty high rate of misclassification, but it is reasonable to presume that most of the misclassification are to classes that are nearby the true class.

5. USING REAL HYPERSPECTRAL AND MULTISPECTRAL DATA

The data-independent experiments provide some guidance with regard to the relative ability of the alternative distance metrics to classify high-dimensional data sets, but the simplified model misses some aspects of a full k-means clustering of a real hyperspectral dataset. For instance, one can presume that most of the misclassifications

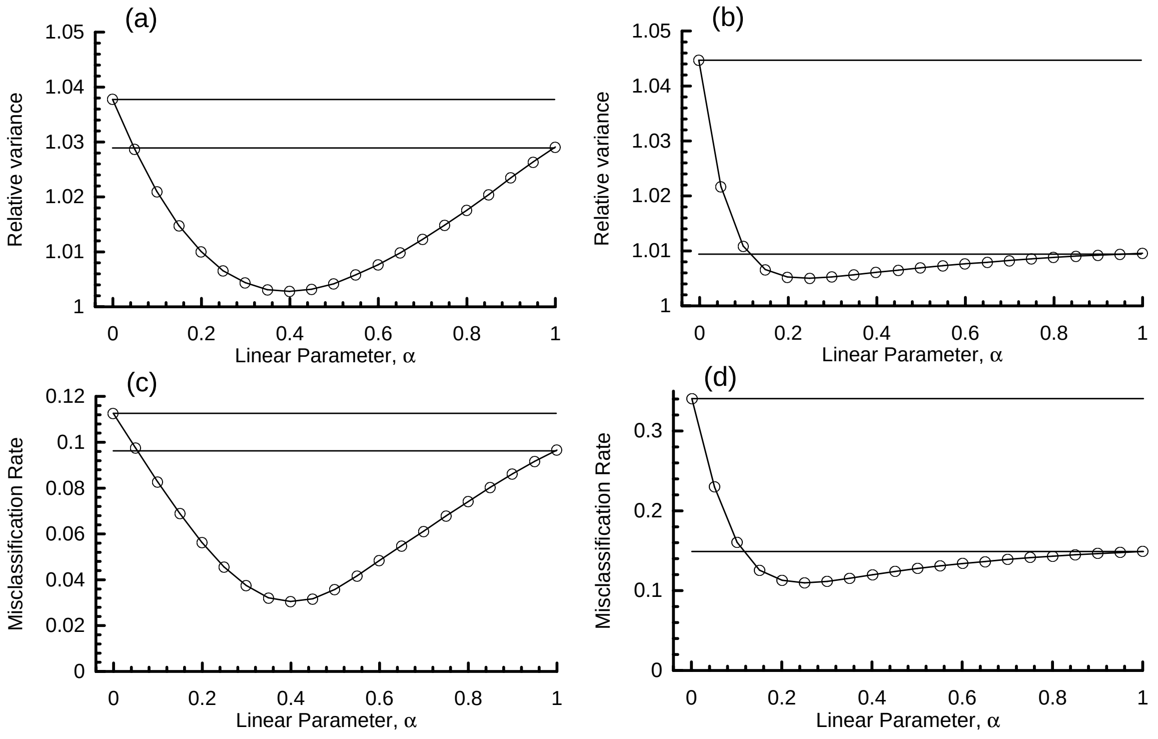


Figure 2. **(a)** This plot shows the relative variance for $K = 2$ classes, plotted against the parameter α of the linear combination of Max and Manhattan metrics (see Eq. (3)), for a low ($D = 3$) dimensional space. **(b)** Same as (a), but with a high ($D = 50$) dimensional space. The horizontal lines correspond to pure Max (upper line, corresponding to $\alpha = 0$) and pure Manhattan (lower line, corresponding to $\alpha = 1$). For the smaller value of D, the optimal linear combination provides a sizeable reduction in the relative variance, compared to the Manhattan and the Max metrics by themselves. For larger D, the linear combination is only slightly better than the Manhattan metric. The optimal value of α decreases, slowly, with increasing D; for this range of D, a value of $\alpha = 0.25$ works pretty well. **(c)** The probability of misclassification using the alternative metrics is shown as a function of α for $D = 3$. **(d)** Same as (c), but with $D = 50$. Particularly for this larger dimension, a small relative variance (less than a percent difference between the Euclidean and the $\alpha = 0.25$ alternative) can lead to a large misclassification rate (near ten percent).

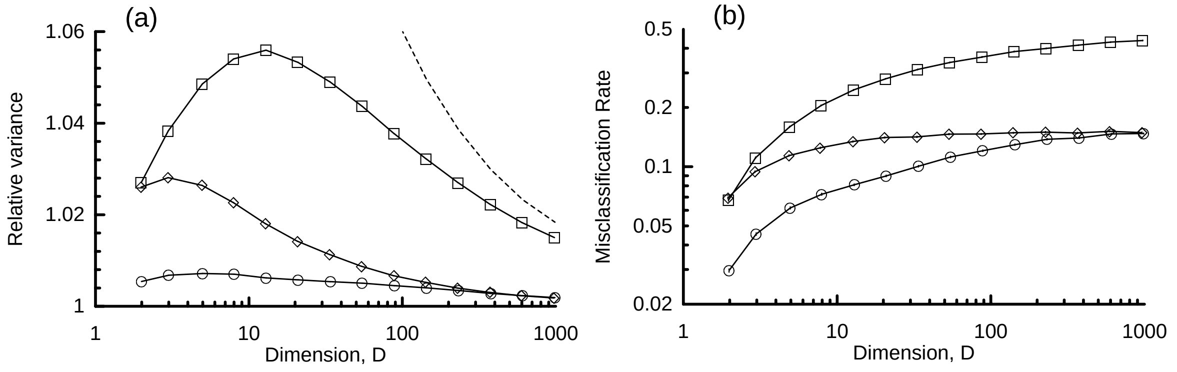


Figure 3. **(a)** Relative variance for $K = 2$ classes, plotted against the dimension D of the sphere onto which a triplet of points have been randomly placed. Two of the points in each triplet represent cluster centers, and the third a random pixel. The diamonds represent the Manhattan metric, the squares represent the Max metric, and the circles correspond to the linear combination with $\alpha = 0.25$. The dashed line indicates the relative variance associated with random assignment of pixels to centers. This is a "worst case" bound for the relative variance. **(b)** Probability of misclassification for the same parameters as in (a). A misclassification occurs when the center that is closest according to the Euclidean metric is not closest according to one of the other metrics. Note that although the relative variance is decreasing for large values of D, the rate of misclassification is monotonically increasing.

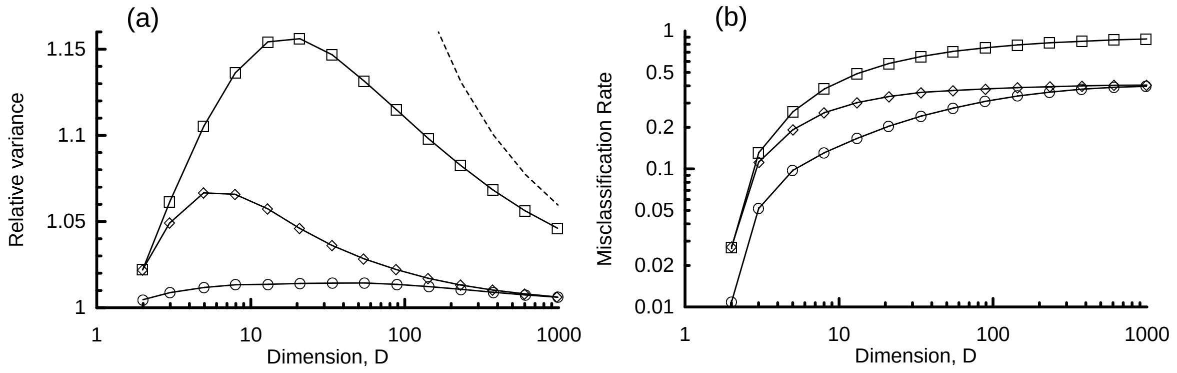


Figure 4. Same as Fig. 3, but for $K = 16$ classes. **(a)** Relative variance plotted against the number D of spectral channels; and **(b)** misclassification rate.

occur when points are nearly equidistant between two candidate centers. One imagines that the process of clustering nonuniform data preferentially produces situations where there are gaps between the clusters, which might reduce the actual rate of misclassifications. Also, although hyperspectral data has nominally a large dimension D, in practice, most of the variance is concentrated in just a few of these dimensions, which effectively reduces the dimensionality of the data. On the other hand, the alternative distance metrics are not rotationally invariant, so it is not clear that they would be able to exploit this lower effective dimensionality.

To compare these metrics in a more realistic situation, we produced a k-means clustering algorithm for which the distance metric was adjustable, and applied it to a set of five AVIRIS data sets. Each data set is a $614{\times}512$ image with 224 channels, spanning the range from 0.4 to 2.5 microns. We also used these AVIRIS data sets to produce associated 10 channel data sets simulating the data produced by the MTI sensor.[5]

We initialized the clustering by dividing the image into K equal width "stripes" so that the first N/K points comprise the first cluster, the next N/K points comprise the second cluster, *etc.* We also considered several other initialization strategies, but this one has the advantage that it is simple and deterministic.

Each image was classified with each of the distance measures, first into $K = 2$ clusters, and then into $K = 16$ clusters. Results are shown in Table 1 and Table 2, with discussion in the table captions.

6. CONCLUSIONS

Using both artificial models and real data, we compared the use of the standard Euclidean distance to several cheaper alternative distance metrics for k-means clustering. Although the Euclidean metric is theoretically optimal in terms of the cluster quality, it is also expensive to compute, particularly in a hardware implementation. We considered three alternative distance metrics (Manhattan, Max, and a linear combination of the two) which would be more amenable to hardware implementation. Both in the data-independent Monte-Carlo studies, and in the comparisons using hyperspectral and multispectral image data, we found that the alternative distances generally produced lower quality clusters than those based on Euclidean distance. For the AVIRIS and simulated MTI data, no consistent preference for Manhattan over Max could be concluded, but the Manhattan did have a lower rate of misclassification, and a smaller relative variance increase, in the idealized Monte-Carlo experiments. The linear combination proved better than the Manhattan and Max metrics in the idealized experiments; in practical cases, it was also generally (but not exclusively) better than the other alternatives.

ACKNOWLEDGMENTS

We are grateful to the MTI team for providing both the AVIRIS and the simulated MTI data. We also thank the other members of the ASAPP team for many useful discussions.

REFERENCES

1. NASA, 1999. `http://makalu.jpl.nasa.gov/avaris.html`.
2. V. Castelli and T. M. Cover, "On the exponential value of labeled samples," *Pattern Recognition Lett.* **16**, pp. 105–111, 1995.
3. P.-F. Hsieh and D. Landgrebe, "Statistics enhancement in hyperspectral data analysis using spectral-spatial labeling, the EM algorithm, and the leave-one-out covariance estimator," *Proc. SPIE* **3438**, pp. 183–190, 1999.
4. R. A. Schowengerdt, *Techniques for Image Processing and Classification in Remote Sensing*, Academic Press, Orlando, 1983.
5. P. G. Weber, B. C. Brock, A. J. Garrett, B. W. Smith, C. C. Borel, W. B. Clodius, S. C. Bender, R. R. Kay, and M. L. Decker, "MTI Mission Overview," *Proc. SPIE* **3753**, pp. 340–346, 1999.
6. A. E. Filip, "Linear approximation to $\sqrt{x^2 + y^2}$ having equiribble error characteristics," *IEEE Trans. Audio and Electroacoustics* **21**, pp. 554–556, 1973.
7. A. E. Filip, "A baker's dozen magnitude approximations and their detection statistics," *IEEE Trans. Aerospace and Electronic Systems* **12**, pp. 86–89, 1976.

Image	Within-class Variance				Fractional Difference from Euclidean		
	Euclidean	Manhattan	Max	$\alpha = 0.25$	Manhattan	Max	$\alpha = 0.25$
f960323t01p02_r04_sc01.c.img	0.3227	0.3241	0.3316	0.3239	0.0112	0.0313	0.0104
f970410t01p02_r02_sc02.c.img	0.3959	0.4001	0.3978	0.3996	0.0054	0.0043	0.0049
f970620t01p02_r03_sc02.c.img	0.5265	0.5324	0.5443	0.5319	0.0181	0.0284	0.0170
f970701t01p02_r07_sc01.c.img	0.7334	0.7970	0.7588	0.7959	0.5867	0.1100	0.5905
f970801t01p02_r01_sc01.c.img	0.6331	0.6402	0.6442	0.6386	0.0419	0.0375	0.0350
f960323t01p02_r04_sc01.c.img.mti	0.3148	0.3169	0.3155	0.3152	0.0212	0.0067	0.0096
f970410t01p02_r02_sc02.c.img.mti	0.4087	0.4106	0.4100	0.4088	0.0044	0.0039	0.0010
f970620t01p02_r03_sc02.c.img.mti	0.5344	0.5366	0.5421	0.5347	0.0145	0.0208	0.0045
f970701t01p02_r07_sc01.c.img.mti	0.7178	0.7675	0.7425	0.6999	0.0713	0.6870	0.1377
f970801t01p02_r01_sc01.c.img.mti	0.6497	0.6698	0.7307	0.6593	0.0454	0.2696	0.0232

Table 1. Comparisons of within-class variance for clusterings of 224-channel AVIRIS data (top five lines) and 10-channel simulated MTI data (bottom five lines), using different distance metrics. These variance measurements are divided by the total variance in the data set. $K = 2$ clusters were used, and the algorithm was run for 10 iterations, or until convergence was achieved. In almost all cases, the Euclidean clustering produced the minimum within-class variance (the single exception was the $\alpha = 0.25$ clustering for data set `f970701t01p02_r07_sc01.c.img.mti`). In all cases, the $\alpha = 0.25$ clustering produced smaller variance than the Manhattan metric, though in most cases the difference was small. Although the Manhattan metric is uniformly superior to the Max metric in the idealized Monte-Carlo studies, the comparison for multiple iterations on real data leads to mixed results; the Max was actually better in two of the five AVIRIS images. The difference from the Euclidean clustering is the fraction of pixels which are labeled differently in the Euclidean clustering and in the clustering produced by the alternative distance. This provides an informal measure of "misclassification" but one should be cautious about interpreting that as a measure of goodness. The comparison is with the clustering produced by the Euclidean distance, not with any "ground truth." Even though the algorithm is deterministic and even though each run started with the same initial conditions, small perturbations caused by the use of a different distance metric, can lead after multiple iterations to quite different clusterings.

Image	Within-class Variance			
	Euclidean	Manhattan	Max	$\alpha = 0.25$
f960323t01p02_r04_sc01.c.img	0.0481	0.0738	0.0523	0.0733
f970410t01p02_r02_sc02.c.img	0.0883	0.0878	0.1030	0.0870
f970620t01p02_r03_sc02.c.img	0.0997	0.1154	0.1141	0.1120
f970701t01p02_r07_sc01.c.img	0.1263	0.1325	0.1725	0.1315
f970801t01p02_r01_sc01.c.img	0.0963	0.0958	0.1047	0.0947
f960323t01p02_r04_sc01.c.img.mti	0.0411	0.0424	0.0427	0.0406
f970410t01p02_r02_sc02.c.img.mti	0.0808	0.0826	0.0798	0.0806
f970620t01p02_r03_sc02.c.img.mti	0.0862	0.0957	0.0947	0.0864
f970701t01p02_r07_sc01.c.img.mti	0.1042	0.1001	0.1032	0.1066
f970801t01p02_r01_sc01.c.img.mti	0.0792	0.0789	0.0884	0.0792

Table 2. Comparisons of within-class variance for clusterings of 224-channel AVIRIS data (top five lines) and 10-channel simulated MTI data (bottom five lines), using different distance metrics. These variance measurements are divided by the total variance in the data set. $K = 16$ clusters were used, and the algorithm was run for 50 iterations, or until convergence was achieved.

Early Experience with a Hybrid Processor: K-Means Clustering

Maya Gokhale, Jan Frigo
Kevin McCabe, James Theiler
NIS-3, NIS-4, NIS-2
Los Alamos National Laboratory
Los Alamos, NM, U.S.A.

Dominique Lavenier
IRISA - CNRS
Campus de Beaulieu
35042 Rennes cedex - FRANCE

Abstract *We discuss hardware/software co-processing on a hybrid processor for a compute- and data-intensive hyper-spectral imaging algorithm, K-Means Clustering. The experiments are performed on the Altera Excalibur board using the soft IP core 32-bit NIOS RISC processor. In our experiments, we compare performance of the sequential algorithm with two different accelerated versions. We consider granularity and synchronization issues when mapping an algorithm to a hybrid processor. Our results show that on the Excalibur NIOS, a 15% speedup can be achieved over the sequential algorithm on images with 8 spectral bands where the pixels are divided into 8 categories. Speedup is limitd by the communication cost of transferring data from external memory through the NIOS processor to the customized circuits. Our results indicate that future hybrid processors must either (1) have a clock rate 10X the speed of the configurable logic circuits or (2) include dual port memories that both the processor and configurable logic can access. If either of these conditions is met, the hybrid processor will show a factor of 10 speedup over the sequential algorithm. Such systems will combine the convenience of conventional processors with the speed of configurable logic.*

Keywords: configurable system on a chip, CSOC, Excalibur, K-means Clustering, image processing

1 Introduction

Over the past ten years, it has been well documented that configurable logic processors composed of SRAM-based Field Programmable Gate Arrays (FPGAs) can accelerate compute-intensive operations by one to two orders of magnitude over Pentium-class processors. However, as more experience has been gained with FPGA processing, it has also become evident that there is much more to any algorithm than a compute-intensive core. File I/O, outer loop management, and other housekeeping tasks make up the bulk of the source code. It is time-consuming to map these functions onto hardware and usually not profitable in terms of speedup - it is better to use hardware to unroll an inner loop for the maximum data flow rather than to map complex control and I/O functions onto hardware.

However, the architecture of currently available FPGA computing platforms does not lend itself easily to hardware/software co-processing. FPGA boards typically communicate with a processor via an I/O bus such as PCI or VME. Not only is the I/O bandwidth between hardware and software slow and pin-limited, but the system overhead to set up a transaction between the processor and FPGA board is high. All these factors dictate that as much of the computation as possible occur in hardware, and that the granularity of transaction between hardware and software is both large and deterministic (so that operations can be scheduled), with minimal synchronization between the two.

Recently, hybrid Configurable System on a Chip (CSOC) architectures, proposed several years ago ([8], [5], [9]), have begun to appear as commercial offerings ([1], [10]). In contrast to traditional FPGAs, these integrated systems offer a processor and an array of configurable logic cells on a single chip. On such systems, it becomes feasible to have software and hardware communicate at clock cycle latency rather than over a slow I/O bus, speeding up synchronization between the two. As a result, a smaller granularity of operation should be possible in hardware as compared to the conventional FPGA board co-processor.

As hybrid processors are still not readily available, there has been to date little experience with mapping algorithms to these devices and measuring performance. In this paper, we present practical experience with using the Excalibur NIOS system for a compute- and data-intensive application in remote sensing, the K-Means Clustering algorithm. We choose this algorithm because it is readily parallelizable in a variety of ways, and FPGA-based acceleration of K-Means kernel loops has previously been reported [7]. We experiment with mapping K-Means to a hybrid processor and evaluate performance of two different mapping techniques.

2 K-Means Clustering

The basic principle of image clustering is to take an original image and to represent the same image using only a small number of pixel values. The goal of the K-Means algorithm is to assign each pixel to one of a pre-defined number NB_CLASS of classes. The assignment is done by minimizing a cost function over the set of NB_CLASS class centers, where the class center is simply the average of pixel values currently assigned to the class. This iterative algorithm compares each pixel to each class center, finds the class center with minimal distance to the pixel, and then assigns the pixel to that class. The algorithm may be run for a fixed number of iterations or until no pixel has moved to a new class.

The K-Means algorithm is typically applied to multi- and hyper-spectral remote sensing imagery. Figure 1 shows such an image.

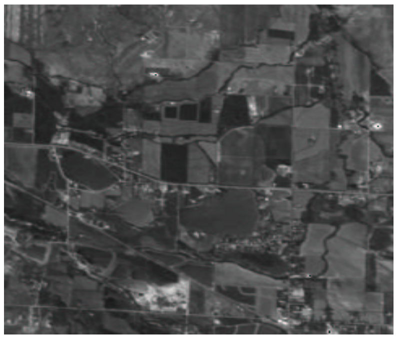


Figure 1: A Multi-Spectral Image

```
Assign pixels randomly to NB_CLASS classes
Compute the centers of the classes
Loop(N) For each pixel,
    • Let C = class of the pixel
    • Determine the class number K which has
      the minimum distance to C
    • if C is not equal to K, move pixel C to
      Class K
Recompute the centers of the classes K and C
```

Figure 2: K-Means Clustering Algorithm

In a multi- or hyper-spectral image, each "pixel" is actually a "hyper-pixel," a vector with a component for each spectral channel in the image. A representative hyper-spectral image might contain 512 x 512 hyper-pixels, where each hyper-pixel is a vector of length 224, and each vector component is 8 – 14 bits long.

Figure 2 outlines the K-Means algorithm, and Figure 3 shows the main K-Means loop in C.

A loop iteration scans all the pixels. For each pixel we check if it still belongs to its class. If not, the pixel is moved to another class and the two centers corresponding to both the new and the old classes are updated. The number of pixels in a class is stored as well as the sum ac-

```
1  while (pixel_move !=0) {
2  pixel_move = 0;
3    for (i=0; i<NB_PIXELS; i=i+B) {
4      for (b=0; b<B; b++) {
5        min = MAX_INT;
6 /* compute distance: pixel <=> all classes  */
7        for (k=0; k<NB_CLASS; k++) {
8          if (N_CENTER[k]!=0) {
9            dist = 0;
10           for (d=0; d<NB_BANDS; d++)
11             dist = dist +
12                 ABS (PIXEL[i+b][d] - CENTER[k][d]);
13           /* find min dist and associated class# */
14           if (x<min) { min = dist; idx[b] = k; }
15         }
16       }
17     }
18     for (k=0; k<NB_CLASS; k++) change[k] = false;
19     for (b=0; b<B; b++) {
20       if (CLASS[i+b]!=idx[b]) {
21         pixel_move ++;
22         k = CLASS[i+b]; N_CENTER[k]--;
23         change[k] = true;
24         for (d=0; d<NB_BANDS; d++)
25           ACC[k][d] = ACC[k][d] -
26                       PIXEL[i+b][d];
27         k = idx[b]; CLASS[i+b] = k; N_CENTER[k]++;
28         change[k] = true;
29         for (d=0; d<NB_BANDS; d++)
30           ACC[k][d] = ACC[k][d] + PIXEL[i+b][d];
31       }
32     }
33     for (k=0; k<NB_CLASS; k++)
34      /* recompute centers if needed */
35       if (N_CENTER[k]!=0 && change[k]==true) {
36         for (d=0; d<NB_BANDS; d++)
37          CENTER[k][d] = ACC[k][d]/N_CENTER[k];
38       }
39   }
40 }
```

Figure 3: K-Means C Code

cumulation necessary for recomputing the class centers. In our implementation, the class centers are periodically updated every block of B pixels. The cost function is an approximation described in [2] well suited to our data set and is computed as the absolute value of a difference. This cost function is well suited to today's configurable hardware. In software, the squared difference is usually used.

The computation can be split roughly into three parts: the distance calculation between a pixel and a class center, the accumulator update, and the center update. In [6], we report the results of profiling the K-Means algorithm. We have found that the most time consuming computation is the distance calculation that compares each pixel value to each class center (see lines 3 – 16 in Figure 3). In the case of 32 classes, this loop consumes more than 99.6% of the computation time. Thus, this calculation is the natural candidate for acceleration.

There are many ways to accelerate K-Means on configurable logic. Two different acceleration approaches have been reported in [7] and [6] (see [3] for a summary of[6]). Both methods put the distance calculation (line 11 of Figure 3) in hardware. [7] pre-loads the image into local memory on the FPGA board and performs all computation except the final center mean calculation in hardware. Thus the entire image must fit in local memory. [6] streams the image pixels through board, and performs only the distance calculation in hardware. It can handle arbitrary size images and scales well to a large number of classes. It incurs communication overhead in repeatedly streaming the image from the processor to the hardware.

3 Mapping K-Means onto a Hybrid Processor

Our hybrid processor model is shown in Figure 4. There is a RISC processor with a variable number and size of busses connecting it to configurable logic. The RISC processor and configurable logic share memory. The configurable logic consists of a "sea of gates" along

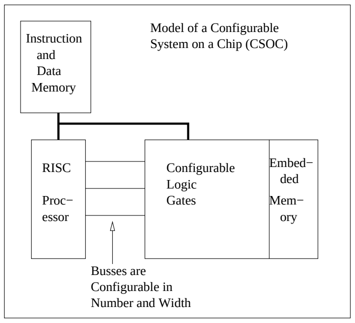

Model of a Configurable System on a Chip (CSOC)
Instruction and Data Memory
RISC Proc–essor
Configurable Logic Gates
Embed–ded Mem–ory
Busses are Configurable in Number and Width

Figure 4: Abstract Hybrid Processor Architecture

with a collection of small embedded memory modules. We refer to a hardware design in the configurable logic as the "user logic." We assume that the processor and user logic run at the same clock speed and that a word may be transmitted between processor and user logic in one clock cycle. The NIOS Excalibur approximates this model, with some important differences. On the NIOS, the user logic cannot access the Instruction and Data SRAM directly. While theoretically the user logic and processor can exchange data in one clock cycle, in reality we measure O(10) clock cycles to send a single 32-bit number from NIOS processor to user logic (see Section 3.1 below).

3.1 Iteration 1: Speeding up Distance Calculation

We approach the problem of mapping K-Means to a hybrid processor incrementally. Since the most time consuming operation is the distance calculation loop, we first map the kernel of that loop to hardware, with all the other code remaining in software. This highlights one of the important advantages of a hybrid processor (see [4] for a more detailed discussion of this point), namely that it is easy with such an architecture to incrementally insert hardware acceleration into a conventional program. We replace a single statement in the C program with a call to the configurable logic. The hardware is a combinational logic circuit with input ports consisting of the distance, the current pixel and current center. The circuit performs the indicated subtraction, abs function and accumulation and returns the updated variable `dist`. Figure 5 shows both the modified C code and the VHDL for this version of the algorithm.[1]

In this example, lines 11 and 12 of Figure 3 are replaced by calls to send the data to the configurable logic and to retrieve the result. The data is sent and received through a set of user-defined busses (see lines 41 – 45).

This hardware logic takes less than 1% of the chip and does not affect the clock frequency of the chip. On the Excalibur, the 32-bit NIOS plus the user logic occupy 22% of the chip, and the clock frequency (fMax) is 31.71. Since we have previously noted that the distance calculation by far dominates the computation time, we might expect the hardware acceleration of this key computation to significantly speed up the K-Means run time. There are two subtracts and one add in the distance calculation. The RISC processor takes at least one clock cycle to execute each of these instructions. All three are done in one clock cycle in the user logic.

In this experiment, the sequential and "accelerated" versions were roughly the same speed. For 64 pixels, with 8 classes and 8 bands, the accelerated version was 15% faster than sequential. When the number of pixels was increased to 224 with 224 classes, the sequential algorithm was 5% faster. This is due to a combination of factors. First, although the arithmetic operations (subtracts and an add) have been accelerated, we have added a cost by communicating the distance, center, and pixel values to the user logic and reading back the updated distance. As the amount of data to be sent to the user logic is increased, the communication overhead begins to dominate the run time.

In an experiment to quantify the cost of sending a single 32-bit value from processor to user logic, we determined that on the Excalibur

[1]Although the send and receive are shown as separate function calls, the calls were manually inlined when measuring speed.

Modified C Code:

```
11 send_data(0,CENTERS,dist,PIXELS);
12 dist = get_result();

...

41 EP_PIO *dist_out = (EP_PIO *) NA_dist_out;
42 EP_PIO *ul_reset = (EP_PIO *) NA_Reset;
43 EP_PIO *center = (EP_PIO *) NA_center;
44 EP_PIO *dist_in = (EP_PIO *) NA_dist_in;
45 EP_PIO *pixel = (EP_PIO *) NA_pixel;
46
47 int send_data(rst, cent, din, pix)
48 int rst, cent, din, pix;
49 {
50   ul_reset->EPR_PIOData = rst;
51   center->EPR_PIOData = cent;
52   dist_in->EPR_PIOData = din;
53   pixel->EPR_PIOData = pix;
54 }
55
56 int get_result()
57 {
58   return (dist_out->EPR_PIOData);
59 }
```

VHDL for Distance Calculation:

```
  dist_process: process(Clk, Reset)
    variable p_i: integer;
    variable c_i : integer;
    variable d_i : integer;
  begin
    if (Reset = '1') then
       dist_out <= "0000000000000000";
    elsif rising_edge(Clk) then
        p_i := conv_integer(pixel);
        c_i := conv_integer(center);
        d_i := conv_integer(dist_in);
        if pixel>center then
          dist_out <=
           conv_std_logic_vector(d_i + (p_i - c_i),16);
        else
          dist_out <=
           conv_std_logic_vector(d_i + (c_i - p_i),16);
        end if;
    end if;
  end process;
```

Figure 5: Hardware Acceleration of Distance Calculation

with a 32-bit NIOS processor, it takes 11 clock cycles[2] to send one 32-bit value from processor to user logic using memory-mapped I/O, which is a 12MB/s rate assuming a 33MHz clock for both processor and user logic. This communication cost more than offsets the gain of performing multiple arithmetic operations in parallel. Second, even if we could communicate a word between processor and user logic in a single user logic clock cycle by increasing the processor clock speed by a factor of 10, there is still a significant amount of address calculation code in the innermost loop that is performed sequentially. Thus the fraction of parallel code relative to the amount of sequential code is quite small, which, by Amdahl's Law, is a limiting factor to speedup.

Our conclusion from this experiment is that communication cost continues to be critical to the granularity of the custom instruction. The speed of communication can be increased by increasing the clock speed of the processor relative to the user logic.

3.2 Iteration 2: Parallelizing across Classes

Our second approach focuses on larger granularity with a more global parallelization of the distance calculation by unrolling the loop over all the classes on lines 7–16. The idea is to flow the pixel stream through a linear array of cells, where the number of cells is equal to the number of classes. A cell k computes the distance between its class k and the current flowing pixel. It also updates the current "best" class that has been found for each pixel (i.e., the class with minimum distance to the pixel). The new class computed for the pixel is returned to the processor, and new class centers are computed. Periodically, a new set of centers is streamed to the array of cells. The cell array is shown in Figure 6.

Each processor has a small memory storing the class center (a vector of NB_BAND values), and performs the following computation:

[2]This includes 2 wait states.

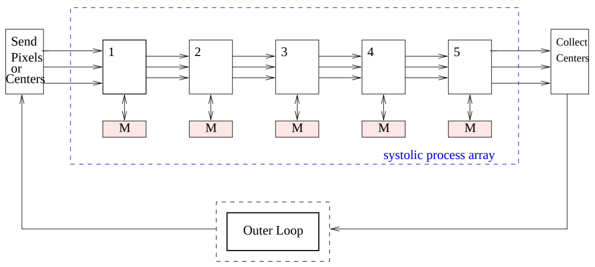


Figure 6: Linear Array Implementation

```
index = my_processor_number;
while (! end_of_stream) {
  dist = 0;
  for (d=0; d<NB_BAND; d++) {
    stream_read (pixel);
    dist = dist + ABS(pixel - center[d]);
    stream_write (pixel);
  }
  stream_read (left_dist, left_index);
  if (dist < left_dist) {
    left_dist = dist; left_index = index;
  }
  stream_write (left_dist, left_index);
}
```

The input data (pixels and class centers) are written from the processor to the user logic through a set of user-defined busses, similar to the method shown in Figure 5.

In this experiment, the accelerated version showed an 11% speedup over the sequential algorithm. Once again, the cost of communicating the image to the user logic, at 11 cycles per word, was the dominating factor that prevented greater speedup. Although there was greater parallel activity in this version than either the sequential version or our first iteration, the high cost of sending the pixel array to the user logic was the limiting factor.

We have compiled this hardware version of the K-Means distance calculation (implemented with 32 classes or cells) for both the Altera APEX20K200 as well as the Xilinx Virtex. The table in Figure 7 shows the size of the user logic on the Altera chip as well two Virtex chips. Each cell holds 224 spectral channels for the class center it represents.

FPGA	Gates	%Usage	Speed
APEX20K200	526K	53%	50 MHz
Virtex V400	468K	70%	25 MHz
Virtex V1000	1.124M	28%	41 MHz

Figure 7: Comparison of K-Means Hardware on Altera and Xilinx (implemented with 32 Classes)

4 Conclusions

We have demonstrated the mapping of a data- and compute-intensive algorithm, K-Means Clustering to a hybrid processor consisting of a RISC processor augmented with configurable logic. We have experimented with two approaches to accelerating the K-Means inner loop with maximum speedup achieved of 15%.

One conclusion we draw from this experiment is that speedup can only be gained by the P-RISC approach of substituting hardware for short segments of sequential instructions if there is very fast communication between processor and user logic. This is especially true if the cost of reconfiguration is factored in, which we did not consider in this experiment.

The higher pay-off approach is to parallelize in hardware at the loop level and to put overhead operations such as address calculation on the hardware. The overhead of communicating data between the processor and user logic remains the primary impediment to higher speedup. Limitations of the clock speed of the soft processor core and memory architecture of the Excalibur NIOS development board are responsible for the measured overhead.

These factors can be remedied by

- using a hard IP core processor that is clocked at 10X the clock rate of the user logic
- separate data and instruction memory
- instruction cache
- dual ported memory shared by processor and configurable logic

- multiple memory banks.

With the dual ported memory, it would be possible to pass to the user logic simply the addresses of the pixel and center arrays, and let the hardware perform pipelined fetch of the pixel data directly. Center update could still be done by the software, with the new class centers written directly to the shared memory. With this approach it is possible for the hybrid chip to deliver one to two orders of magnitude speedup over software.

Extrapolating to the Virtex 1000 PowerPC, we can fit 96 classes. If the memory communication can proceed at 40MB/sec (comparable to 32-bit PCI DMA mode), a speedup of approximately 200 [6] can be realized over the software version. This communication rate can either be achieved by running the processor at 400MHz or by providing a path for the user logic to access memory concurrently with the processor.

Thus, despite the modest measured results from the Excalibur NIOS, we believe it is possible for the performance of a hybrid processor to be orders of magnitude faster than a conventional processor. However, careful attention to the system architecture is necessary to realize these benefits.

Acknowldegements: We are grateful to Konstantin Borozdin for helping compile and debug version 1 of the K-Means to the Excalibur.

References

[1] Altera Corporation. Excalibur. *http://www.altera.com/products/devices/excalibur/exc-index.html*, 2001.

[2] M. Estlick, M. Leeser, J. Szymanski, and J. Theiler. Algorithmic Transformations in the Implementation of K-means Clustering on Reconfigurable Hardware. *ACM FPGA 2001*, 2001.

[3] J. Frigo, M. Gokhale, and D. Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective. *ACM FPGA 2001*, 2001.

[4] M. B. Gokhale and J. M. Stone. Co-synthesis to a hybrid RISC/FPGA architecture. *Journal of VLSI Signal Processing Systems*, 24, March 2000.

[5] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, Apr. 1997. To be published.

[6] D. Lavenier. FPGA Implementation of the K-Means Clustering Algorithm for Hyperspectral Images. *Los Alamos National Laboratory LAUR 00-3079*, 2000.

[7] M. Leeser, M. Estlick, N. Kitaryeva, J. Theiler, and J.Szymanski. Applying Reconfigurable Hardware to Segmentation for Multispectral Imagery. In *HPEC 2000*, Boston, MA, Sept. 2000.

[8] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, Nov. 1994.

[9] C. Rupp et al. The Napa Adaptive Processing Architecture. *FCCM 1998*, Apr. 1998.

[10] Xilinx Corporation. Virtex/powerpc. *http://www.xilinx.com/prs_rls/ibmpartner.htm*, 2000.

Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective

Jan Frigo
Los Alamos National Laboratory
Los Alamos, NM, 87545
jfrigo@lanl.gov

Maya Gokhale
Los Alamos National Laboratory
Los Alamos, NM, 87545
maya@lanl.gov

Dominique Lavenier
IRISA - CNRS
Campus de Beaulieu
35042 Rennes cedex - FRANCE
lavenier@irisa.fr

ABSTRACT

The Streams-C compiler ([5]) synthesizes hardware circuits for reconfigurable FPGA-based computers from parallel C programs. The Streams-C language consists of a small number of libraries and intrinsic functions added to a synthesizable subset of C, and supports a communicating process programming model. The processes may be either software or hardware processes, and the compiler manages communication among the processes transparently to the programmer. For the hardware processes, the compiler generates Register-Transfer-Level (RTL) VHDL, targeting multiple FPGAs with dedicated memories. For the software processes, a multi-threaded software program is generated.

The Streams-C language and compiler offer a very high level of expressivity for reconfigurable computing application development, particularly for stream-processing applications. We find this is reflected in productivity, for a factor of up to 10 times improvement in time to produce a program. However, use of the tool in the "real world" is predicated on performance: only if such a compiler can deliver performance comparable to hand-coded performance will it be used in practice.

This paper presents an application study of the Streams-C compiler. Four applications have been written in Streams-C and compiled to the AMC Wildforce board containing Xilinx 4036's. Those same applications have been hand-coded in a combination of RTL and structural VHDL. We compare performance of the generated code with the hand-optimized code. Our study shows that the compiler-generated designs are 1.37–4 times the area and 1/2–1 times the clock frequency of the hand designs. We find that the compiler, based on the SUIF infrastructure, can be greatly improved through various standard compiler optimizations that are not currently being exploited. Thus we are currently rewriting a public domain version of Streams-C to better optimize and target the Virtex chip.

Keywords

FPGA, configurable computing, silicon compiler, FPGA design tools, high-level synthesis, hardware-software co-design

1. INTRODUCTION

Over the past ten years, Reconfigurable Computing has demonstrated factors of 10 to 100 speedup over conventional high performance workstations at relatively modest cost. Field Programmable Gate Array (FPGA)-based accelerator boards with customized hardware programmed into the FPGAs have been used in signal and image processing applications for real-time embedded computation.

A major drawback to the widespread use of Reconfigurable Computing has been the cost of developing applications for these parallel systems. Current state of practice is to use low level Hardware Description Language (HDL) to describe the circuits realizing an algorithm. Not only is this task extremely time consuming, but it requires hardware design expertise. Thus successfully fielding applications for Reconfigurable Computers typically requires a team of domain experts, software programmers, and hardware designers.

There has been considerable research interest in reducing design time for Reconfigurable Computer applications. Approaches have ranged from high-level optimization schemes, to low-level, technology-specific, optimized designs. Some examples of high-level optimization methods include: a C to HDL method for high-speed pipeline circuits, focusing on the exhaustive parts of an application such as loop and recursive programs([10]); and a pipeline vectorizaton technique([15]) to optimize and pipeline candidate inner space loops for hardware acceleration. Other techniques map a MATLAB application([11]) to a distributed computing environment, use graphical programming ([13], [14]), employ automatic parallelization and synthesis ([7]) techniques. Low-level efforts target technology-specific, optimized designs ([2]) or automatic data storage and control ([12]) for computation.

In the Streams-C approach, we target an intermediate level of expression. Our compiler processes a subset of C suitable for automatic synthesis to FPGAs. Our programming model is targeted at stream-oriented reconfigurable computing applications. In this model, parallel processes communicate via data streams. We optimize compiler synthesis for high-rate flow of data streams; small, fixed size data packets; and low-precision fixed point computation, all characteristics of FPGA-based reconfigurable parallel processing applications. Our system includes a functional simulation environment based on POSIX threads, allowing the programmer to simulate the collection of parallel processes and their communication at the functional level. Our compiler currently targets the Annapolis Microsystems Wildforce board, containing 5 Xilinx 4036 FPGAs and four banks of 32bit x 65K SRAM.

In the next section, we briefly review the Streams-C language. Next we describe the performance study methodology, describe each application, and compare performance between compiler-generated versus hand-optimized code. We end with a summary of results and a discussion of future work.

2. OVERVIEW OF STREAMS-C

The concept of stream-based computation is a fundamental formalism for high performance embedded systems, which is characterized by (multiple) streams of data produced at a high rate, with complex operations performed on the incoming data. The Streams-C [5, 3] language supports this computational model with a minimal number of language extensions and library functions callable from a C program. The compiler targets a combination of software and hardware.

For computation occurring in hardware, the compiler generates RTL VHDL for a target FPGA board containing multiple FPGAs, external memories, and interconnect. The language extensions, such as declarations for a process or stream, allocate resources on the board for these objects. These extensions allow the programmer to allocate registers on an FPGA and define register bit lengths; assign variables to memories; define concurrent processes; define stream connections between processes; and read/write streams to communicate data between processes. The processes operate asynchronously, and synchronize through stream operations, which may occur anywhere within the body of the process. A distributed memory model is followed, with local state belonging to each process and inter-process communication via streams. The extensions include mapping directives to give the applications developer control over the mapping of processes to hardware components and of streams to communication media on the target application board.

A hardware streams library has been built for the Annapolis Microsystems Wildforce accelerator board. The compiler, based on the Napa C compiler and Malleable Architecture Generator (MARGE), synthesizes hardware circuits from a C-language program. Although the target is a synchronous set of circuits on multiple communicating FPGAs, the C programmer does not have to be concerned with synchronizing state machines, or other hardware timing events. The compiler-generated state machines control sequencing and loops. The hardware streams library encapsulates the data flow synchronization between stream reader and writer. The combination of compiler-generated computation nodes with the hardware streams library allows applications developers to target FPGA boards from a high level concurrent language.

A software library using POSIX threads provides concurrent processes and stream support in software. Thus the software libraries support a dual function: when all processes are mapped to software, our system provides a functional simulation environment for the parallel program.

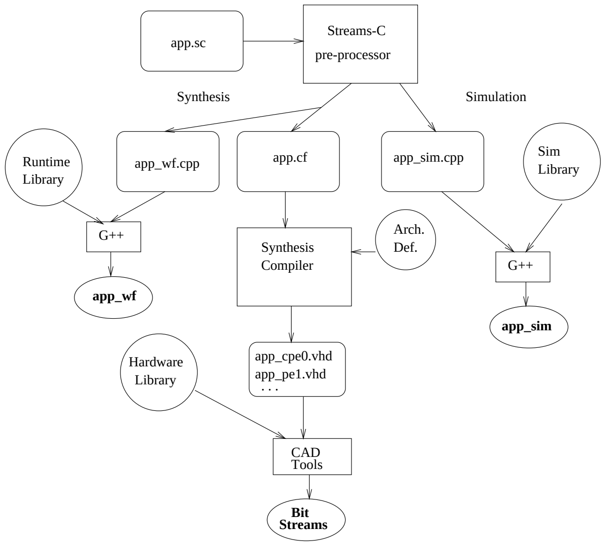


Figure 1: Organization of the Streams-C compiler: The application written in streams-C (app.sc) goes through the streams-C preprocessor which produces three descriptions: app_sim.cpp for simulation purpose, app.cf as an entry point for the synthesis compiler, and app_wf.cpp for running the host process. The synthesis compiler translates the app.cf file into a VHDL description for each processing element of the Wildforce board. Thus the VHDL files are applied to the Xilinx CAD tools which generate the bit-stream for the FPGA components.

When processes are mapped to a combination of software and hardware, the software libraries are used for communication among software processes and between software and hardware processes. Hardware libraries are used for communication among hardware processes and for the hardware side of communication to software processes. Figure 1 shows the software development flow for applications using the Streams-C compiler.

3. APPLICATION STUDY

In this section, we will discuss four applications that have been mapped to reconfigurable hardware. These are

- contrast enhancement (previously reported in [6])
- polyphase filter bank[1]
- Pixel Purity Index (PPI) [8]
- K-means clustering [9]

The contrast enhancement algorithm is used for grayscale adjustments to pixels in an images. The polyphase filter bank[1], is a key component of the digital receiver architecture being developed by Los Alamos (see rcc.lanl.gov). The PPI and K-means clustering perform classification of features in multi- and hyper-spectral imagery.

Each of these applications was implemented with hand-coded VHDL and with the Streams-C compiler generated

VHDL. The same target hardware, the Annapolis Microsystems Wildforce board, was used for each. This board consists of five Xilinx 4000 Series FPGAs ($X0 \dots X4$), each with a dedicated SRAM. $X0, X1$ and $X4$ have bi-directional FIFO connections to the host processor. Each FPGA $X1 \dots X4$ has direct connection to its immediate linear neighbor. In addition, all the FPGAs can communicate over a crossbar.

In this section, the algorithms will be described with respect to algorithm parallelization and mapping to the hardware. We compare hardware area utilization, speed and development time estimates for both versions.

3.1 Contrast Enhancement

Histogram projection contrast enhancement is a well known image processing transformation to perform grayscale adjustment to pixels in an image. Using statistics of the image itself to control adjustments, the algorithm stretches contrast within the image to use the entire dynamic range of the display. It is commonly used in IR video enhancement.

This algorithm was previously reported in [4], and thus we briefly summarize the results here. The algorithm has several phases. First, a histogram of the input image is generated (Phase 1, Histogram Generation) and the total number N of grayscale values in the image is computed. New grayscale values are then assigned, with the darkest grayscale value getting value 0, and the brightest grayscale value getting value N-1. Intermediate brightness pixels are given values in the range 0 through N-1. The new assignments are stored in a "contrast stretch table" (Phase 2, Contrast Stretch Table Generation). Next the image is remapped to the new grayscale values by setting the new grayscale value (n) for each input pixel to be n/N, yielding a scaled value in the 8-bit range (Phase 3, Image Remapping). The new pixel value is then output.

In mapping to the Wildforce, Phases 1 and 2 are performed on a single chip, and Phase 3 on a different chip. Phase 1 reads the input pixel, updates the histogram table, and writes the pixel to memory. When all pixels of the image have been read in, Phase 2 assigns the "stretch" values, and reads the pixels back out of memory, passing the pixel and stretch amount to an adjacent chip. Phase 3 does the divide with a table lookup into its SRAM bank. The partitioning between two chips is driven by the fact that Wildforce only has one memory bank per FPGA chip, and Phase 3 needs a dedicated memory for the table lookup. Thus for maximum parallelism, two chips are used.

Written in Streams-C, this program consists of two host processes and 2 FPGA processes. The first host process reads images from disk and sends four pixels at a time to a "controller" process on P0. The controller simply forwards stream pixels onto the crossbar, which broadcasts the pixel stream to processes on X2, which performs phases 1 and 2. The X2 process then sends an input pixel plus the scaling amount in a 16-bit packet to X1, which does the table lookup and then sends groups of four output pixels to a host process. The host process assembles the output frame.

The most computationally intensive processing is done on X2, the process performing histogram and contrast table generation. For that design, the hand-crafted version used 18 percent of the chip, and runs at 40 MHz. The compiler generated version uses 57 percent of the chip, and runs at 20 MHz. Thus compiler overhead adds a factor of 3 to the area, and a factor of 2 to the clock frequency.

The hand-coded design outputs a result every clock cycle. In the compiler-generated design, a result is output every other clock cycle. This is because the hand-coded version uses the CLB RAMS to store the histogram table, and so can do two memory operations in one cycle (store pixel and store histogram value). While the compiler supports multiple memories and can produce a pipeline schedule with single-tick result generation, we have not yet made CLB RAMs accessible to the compiler, and thus had to sequentialize the memory writes to a single memory.

In terms of design time, the hand done version took a month to get working, while the Streams-C version took a couple of days. This translates to a factor of 10 in productivity.

3.2 Poly-phase Filter Bank

In the field of signal detection, multi-rate filter banks have been employed to help detect RF signals in noisy environments. By decomposing a signal into various frequency subbands, filter banks enhance many algorithms because they make it easier to identify pertinent material on a band by band basis. The polyphase implementation[1] is a multirate filter structure combined with a Fast Fourier Transform (FFT) designed to extract subbands from an input signal[1]. The polyphase filter portion of the structure is based on a prototype baseband lowpass Finite Impulse Response (FIR) filter with symmetric coefficients, i.e., the first $n/2$ and the last $n/2$ coefficents are the same, albeit in reverse order. The remaining filters of the filter bank are frequency shift versions of the prototype. The symmetry of this prototype filter combined with the structured frequency shifts allows for an optimal implementation of the filter bank. First, a prototype low-pass FIR filter, *h0[n]*, with the desired filter parameters is designed. The polyphase filters, *pk[n]*, are expressed in terms of the prototype filter,

pk[n] = *h0[k + lM]* k = 0..*M*-1, $l = 0..L-1$

n is the length of the FIR prototype, M is the number of polyphase filters, L is the length of the individual polyphase filters, (L = *n/M* = 4). The FFT is used following the polyphase filtering structure to provide the frequency shifts for the various channels.

Figure 2 shows the parallelization for two polyphase symmetric filters and below is the Streams-C source code:

```
//coefficients for filter
#define C1 1
#define C2 117
#define C3 1741
#define C4 128

  SC_FLAG(tag);
  SC_REG(data, 32);
  SC_REG(data_o, 32);
  int s; //sample input data
  int in1, in2, in3, in4;
  int e1, e2, e3, e4;
  int o1, o2, o3, o4;
  int evenp; //flag for even or odd data
  int y1, y2; //filter output data

  while(SC_STREAM_EOS(input_stream) != SC_EOS) {
#pragma ALP pipeline
    //Get input data samples from the stream
    s = SC_REG_GET_BITS_INT(data, 0, 8);
```

[1]The filter structure was developed in collaboration with Prof. John Villesenor's team at UCLA.

```
    in1 = C1 * s;
    in2 = C2 * s;
    in3 = C3 * s;
    in4 = C4 * s;

    if (evenp) {
      y1 = e1 + in1; //convolution of even data
      e1 = e2 + in2;
      e2 = e3 + in3;
      e3 = in4;
      SC_REG_SET_BITS_INT(data_o, 0, 16, y1);
    }
    else {
      y2 = o1 + in4; //convolution of odd data
      o1 = o2 + in3;
      o2 = o3 + in2;
      o3 = in4;
      SC_REG_SET_BITS_INT(data_o, 0, 16, y2);
    }
  evenp = !evenp;

    SC_STREAM_WRITE(output_stream, data_o, tag);
    SC_STREAM_READ(input_stream, data, tag);
   }
```

For comparison a filter bank of four is implemented on one chip, computing only the convolution of even data samples.[2] The input comes onto the FPGA via a stream from the host and is unsigned, fixed point, 8 bit data. The coefficents are unsigned, fixed point 12 bit values. The hand-coded design mapped a bank of four polyphase filters to 27% of the area at 40 MHz and the Streams-C version resulted in 37% area utilization for the same speed. We notice that the Streams-C compiler optimized away a multiplier when the coefficient, C1, for the multiplier was one, i.e. Streams-C implemented three multipies in the generated VHDL code. The hand-coded version relies on the synthesis tool, (Synplify in this case) to optimize the multiply operations. Both designs deliver a result every clock cycle. The manual version of the filter took about two weeks to implement on the hardware while Streams-C design-to-implementation took a few days, a development time savings of approximately 5 times.

3.3 Pixel Purity Index

The Pixel Purity Index (PPI) is an algorithm employed in remote sensing for analyzing hyperspectral images. Particularly for low-resolution imagery, a single pixel usually covers several different materials, and its observed spectrum is (to a good approximation) a linear combination of a few *pure* spectral shapes. The PPI algorithm tries to identify these pure spectra by assigning a pixel purity index to each pixel in the image; the spectra for those pixels with a high index value are candidates for basis elements in the image decomposition.

The algorithm proceeds by generating a large number of random D-dimensional vectors, called skewers, through the hyperspectral image. For each skewer, every data point is projected onto the skewer, and the position along the skewer is noted. The data points which correspond to extrema in the direction of a skewer are identified, and placed on a list. As more skewers are generated, this list grows. The number of times a given pixel is placed on this list is also tallied.

[2]The loop as written is pipelinable, but our compiler does not yet pipeline loops containing "if" statements. This extension to the compiler is in progress.

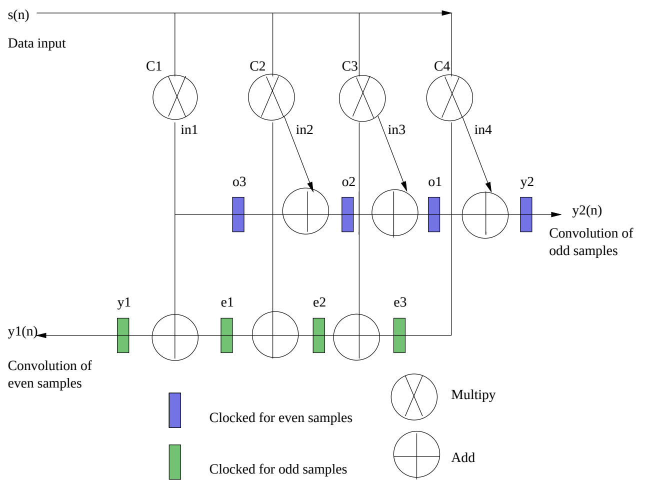


Figure 2: Poly Phase filter bank implementation

The pixels with the highest tallies are considered the most pure, and the pixel's count provides its pixel purity index.

Most of the execution time of the PPI algorithm is spent in computing dot-products between the pixels and the skewers. These dot-product are highly independent and could be done simultaneously. This leads to many ways to parallelize the algorithm, but our approach targets the limited resources available on real FPGA boards. A sequential version of the Pixel Purity Index algorithm [8] is:

```
  PIXELS[N][D]; // an image of N hyperpixels
  SKEWER[K][D]; // a set of K random skewers
  PPI[N];       // the PPI result

// reset pixel purity index
  for (n=0; n < N; n++) PPI[n]=0;
  for (k=0; k < K; k++)  // K skewers
   {
     dpmax=MIN_INT; dpmin=MAX_INT;
     for (n=0; n < N; n++) // N pixels
      {
// compute a Dot-Product
        dp = 0;
        for (d=0; d < D; d++)
           dp = dp + SKEWERS[k][d]*PIXELS[n][d];
// detect extrema
        if (dp > dpmax) { imax=n; dpmax=dp; }
        if (dp < dpmin) { imin=n; dpmin=dp; }
      }

     // update PPI
     PPI[imax]++;
     PPI[imin]++;
   }
```

For each skewer, N dot-products are computed to determine the two pixels which produce the largest and the smallest dot-product. The pixel index (PPI vector) is modified accordingly. A pixel n is a candidate to be a pure pixel if PPI[n] has a high value.

From the above description it can easily be seen that all the dot-products can be computed independently: there are no dependencies between any of them. The parallelization takes advantage of this by computing $KS \times NS$ dot-products simultaneously, where KS and NS represent respectively the number of skewers and pixels which can be processed in

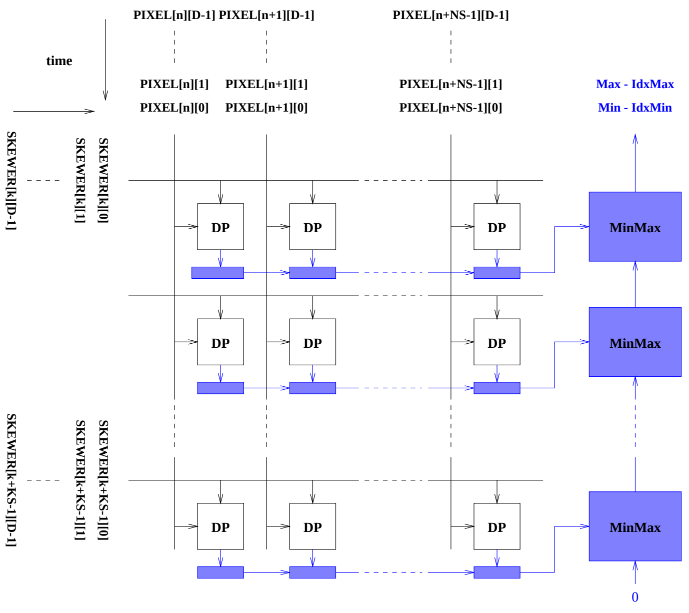


Figure 3: PPI algorithm architeture mapping to the Wildforce board

parallel. This mapping to the hardware is shown in Figure 3.

The skewer data is represented as signed 3 bit data and is input via a stream from the host. The pixel data is unsigned, fixed point, 8 bit data which is stored in off-chip memory. We compare the most computationally intensive part of the algorithm, the dot product (DP), for both versions of this algorithm. The hand-coded version of the dot product maps 2 dot products at 25 MHz with an 22.5% area utilization of the chip after place and route. The Streams-C version has 100% chip area utilization for two dot products at a speed of 15 MHz. Since the main pipelineable loop contains "if" statements, we are not able to pipeline the loop. However, manual application of our extended pipeline algorithm to the loop yields a schedule that delivers an output every clock cycle. The hand-coded algorithm also has this throughput. The hand-coded version was manually placed and the dot product units were manually packed into CLBs. The hand-coded version took six weeks of development time while the Streams-C approach took four to five days, a productivity speed up of 6 times.

3.4 K-means Custering Algorithm

The basic principle of the image clustering process is to take an original image and to represent the same image using only a small number of pixel values[9]. The K-means clustering algorithm performs this task by attempting to minimize a cost function (the absolute value of a difference) over a set of NB_CLASS cluster centers. First, the algorithm assigns pixels randomly to NB_CLASS classes, computes the centers of the classes. There is a outer loop for a number of iterations, N, which can be either fixed in advance or undetermined, and an inner loop which scans all the pixels. For each pixel we check if it still belongs to its class. If not, the pixel is moved to another class and the two centers, corresponding to both the new and the old classes, are updated. The number of pixels in a class is stored as well as the sum

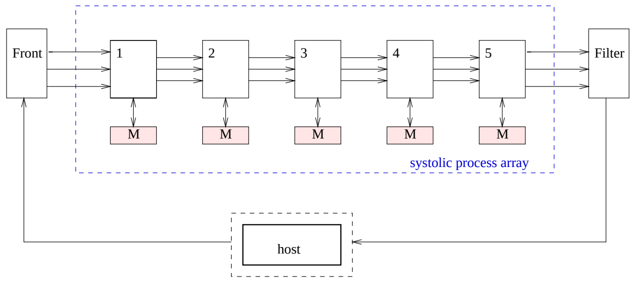


Figure 4: K-means hardware implementation

accumulation necessary for recomputing the class centers. The class centers are periodically updated every block of B pixels.

The computation can roughly be split into three parts: the distance calculation between a pixel and a class center, the accumulator update and the center update. The most time consuming part of the algorithm is the distance computation between the pixels and the class centers, even if the class center is freqently updated. The accumulator and the class center updates represent only a small percentage of the total computation time, especially for a partition into a large number of classes. For example, for a class partition of 32, the distance computation represents more than 99.6 % of the computation time.

Our architecture focuses only on parallelizing the most time consuming part, that is the distance computation between the pixels and the class centers. The idea is to flow a pixel stream through a linear array of processors. The number of processors is equal to the number of classes. A processor k computes a distance between the class k and the current flowing pixel. The result is taken at the rightmost end of the array by the filter process and corresponds to the index class for which a minimum distance has been found. The algorithm mapping to the Wildforce is shown in Figure 4.

Each processor has a small memory storing the class center (a vector of NB_BAND values), and performs the following computation:

```
index = my_processor_number;
while (! end_of_stream) {
  dist = 0;
  for (d=0; d<NB_BAND; d++) {
    stream_read (pixel_value);
    dist = dist + ABS(pixel_value - class_center[d]);
    stream_write (pixel_value);
  }
  stream_read (left_dist, left_index);
  if (dist < left_dist) {
    left_dist = dist; left_index = index;
  }
  stream_write (left_dist, left_index);
}
```

The above code does not compute class centers: it only determines the class number of a pixel. This information is available at the rightmost end of the array each time a pixel (its last vector element) comes out of the array. The host processor is in charge of flushing the pixel stream to the array, getting the results indicating the class number of each pixel, and if a pixel has moved, recalculating the class center accordingly.

The input data (pixel and class centers) comes to the FPGA board from the host as previously described. The Streams-C version implements one processor per this algorithm. It utilizes 14 percent of the area on the chip at a speed of 20 MHz. For a pipelined Streams-C implementation of this algorithm, an output every clock cycle is expected. In comparision, the hand-coded version uses 9.4 percent at the same speed. The Streams-C application took one day of development time, and the hand-coded version took one to two weeks.

4. SUMMARY

The results of place and route with the Xilinx 4036 architecture are shown in Figure 5. The results for the pixel purity index, and the contrast enhancement implementations show that Streams-C generally has a two to three times increase in area utilization on the chip at about one half the clock speed compared to the hand-coded versions. The time savings for implementation is approximately 5 to 10 times in favor of Streams-C. The efficiency of Streams-C compared to hand-coding depends greatly on how the algorithm is parallelized and what operations are mapped to the hardware. For example, the K-means clustering application shows that functions such as addition or subtraction can be automated rather efficiently by Streams-C, without much increase in area utilization or clock speed compared to the manual implementation. The Streams-C productivity increase for the Kmeans implementation is a speed up of 12 times. Key to the success of faster algorithm run-time is assessing the algorithm, parallelizing the computationally expensive processes, and mapping them to hardware. This was accomplished successfully with the Kmeans clustering method by using the host to do part of the processing, and the hardware to do the computationally expensive parts of the algorithm.

Operations such as multiplication are not optimized for synthesis or place and route in this implementation of Streams-C. For this case, manual implementation decreases area utilization and improves speed, for example, in the poly-phase filter application, Streams-C uses 37% of the total area of the chip. Three multipliers are synthesized and one multiplier is optimized away by the compiler, otherwise the area utilization would be almost double that of the hand-coded version. In addition, Streams-C does not yet handle local arrays without accessing off-chip memory which could make a large improvement to area utilization and speed for algorithms like the pixel purity index and contrast enhancement. The continuing Streams-C compiler research will manage local arrays internal to the chip.

The synthesis results for the Virtex 1000 architecture are shown in Figure 6. The Virtex has 27,648 logic cells, 96x64 array of configuration logic blocks (CLBs), the functional elements for constructing user logic. In comparison, the Xilinx 4036E has 3078 logic cells and 1296 CLBs. For technology such as the Virtex series architecture, if your objective is for fast turn around time for alternative implementations of reconfigurable components, Streams-C benefits the user via the development time savings. It is usually not feasible to hand-code alternative implementations without such a tool.

The future version of the Streams-C compiler will handle local, on-chip memory, pipeline loops with control flow, arrays of processes on-chip, and variable length data type declarations, all of which should help optimize the area utilization and speed of an application.

Xilinx 4036 Architecture

	Streams-C VHDL			Handcoded VHDL		
	Area %	Speed MHz	Time wks	Area %	Speed MHz	Time wks
CE	55	20	0.5	18	40	4
PPF	37	40	0.5	27	40	2
PPI DPs 4x2	100	15	1	22.5	25	6
Kmeans	14	20	0.3	9	20	1-2

Figure 5: Place and route results for the Xilinx 4036 on the Wildforce board comparing the Streams-C versus handcoded VHDL.

Virtex V1000 Architecture

	Streams-C VHDL		Handcoded VHDL	
	Area %	Speed MHz	Area %	Speed MHz
CE	3	40	1	40
PPF	1	40	1	40
PPI DPs 4x2	6	40	2	45
Kmeans	< 1	40	< 1	40

Figure 6: Synthesis results for the Virtex X1000 comparing the Streams-C versus the handcoded VHDL .

Reconfigurable Computing is a well known speed up over conventional software system implementations, for example, the results of the Pixel Purity Index (PPI) [8] show the FPGA implementation speed up of 80 times over the software implementaton. The objective of this study is to analyze the performance of the Streams-C compiler with respect to optimized hand-coded designs for practical applications in the image and signal processing domain. Four different applications were choosen, in order to show that the parallelization of the algorithm, and efficient hardware mapping impacts the run-time speed of the application. Streams-C benefits the user via faster development time, but area utilization is the penalty.

This application study shows that C-to-hardware technology in conjunction with a parallel programming model and efficient hardware libraries is within reach for eventual production use. Our future work is directed toward a new Streams-C implementation that will be available as modules within the SUIF compiler framework.[3] In addition to restructuring the compiler phases to better exploit standard optimizations, we plan to add support for arrays of processes and streams and for block and CLB RAMS on a variety of boards targeting the Virtex chip.

5. ACKNOWLEDGEMENTS

The application study was supported by Los Alamos National Laboratory's Deployable Adaptive Computing Systems (DAPS) program. The Streams-C compiler development and the polyphase filter algorithm were primarily supported by the DARPA Adaptive Computing Systems Program. The hand-coding was supported by a DoD grant.

[3](see suif.stanford.edu)

6. REFERENCES

[1] Joseph Arrowood. Comparison of filter banks for signal detection. In *LAUR Number 99-4551*, Los Alamos, NM, March 2000.

[2] Xilinx Corp. http://www.xilinx.com/xilinxonline/jbits.htm. 1999.

[3] M. B. Gokhale, J. Frigo, and J. Stone. Parallel c programming of reconfigurable computers: the Streams-C approach. In *HPEC 2000*, September 2000.

[4] M. B. Gokhale and J. M. Stone. Co-synthesis to a hybrid RISC/FPGA architecture. *Journal of VLSI Signal Processing Systems*, 24, March 2000.

[5] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE international Symposium on FPGAs for Custom Computing Machines*, 2000.

[6] Maya Gokhale, Janice Stone, and Edson Gomersall. Co-synthesis to a hybrid risc/fpga architecture. *Journal of VLSI Signal Processing Systems*, September 2000.

[7] Mary Hall et al. Defacto: A design environment for adaptive computing technology. *Proceedings of the 6th Reconfigurable Architectures Workshop (RAW'99)*, 1999.

[8] Dominique Lavenier, James Theiler, John Szymanski, Maya Gokhale, and Janette Frigo. Fpga implementation of the pixel purity index algorithm. In *SPIE, FPGAs and Reconfigurable Processors for Computing and Applications, vol 4212*, Boston, MA, November 2000.

[9] Miriam Leeser. Applying reconfigurable hardware to segmentation for multispectral imagery. In *HPEC 2000*, Boston, MA, September 2000.

[10] T. Maruyama and T. Hoshino. A c to hdl compiler for pipeline processing on fpgas. In *FCCM 00*, Napa, CA, April 2000.

[11] et. al. P. Banerjee. A matlab compiler for distributed, heterogeneous, reconfigurable computing systems. In *FCCM 00*, Napa, CA, April 2000.

[12] J. Park P. Diniz. Automatic synthesis of data storage and control structures for fpga-based computing engines. In *FCCM 00*, Napa, CA, April 2000.

[13] Eric Pauer, Paul Fiore, John Smith, and Cory Myers. Algorithm analysis and mapping environment for adaptive computing systems. *FPGA2000*, 2000.

[14] S. Periyayacheri et al. Library functions in reconfigurable hardware for matrix and signal processing operations in matlab. *Proc. 11th IASTED Parallel and Distributed Computing and Systems Conference (PDCS'99)*, November 1999.

[15] Markus Weinhardt and Wayne Luk. Pipeline vectorization for reconfigurable systems. In *FCCM 99*, April 1999.

Evolving Network Architectures with Custom Computers for Multi-Spectral Feature Identification

Reid Porter, Maya Gokhale Neal Harvey, Simon Perkins and Cody Young
NIS-2, Space and Remote Sensing Sciences
Los Alamos National Laboratory, NM, 87544
rporter@lanl.gov

Abstract

This paper investigates the design of evolvable FPGA circuits where the design space is severely constrained to an interconnected network of meaningful high-level operators. The specific design domain is image processing, especially pattern recognition in remotely sensed images. Preliminary experiments are reported that compare Neural Networks with a recently introduced variant known as Morphological Networks. A novel network node is then presented that is particularly suited to the problem of pattern recognition in multi-spectral data sets. More specifically, the node can exploit both spectral and spatial information, and implements both feature extraction and classification components of a typical image processing pipeline. Once trained, the network can be applied to large image data sets, or at the sensor to extract features of interest with two orders of magnitude speed-up compared to software implementations.

1. Introduction

Evolutionary Algorithms (EA) are a simple but powerful approach to optimization. The fundamental problem with EA is computation time. Field Programmable Gate Array (FPGA) implementations can provide significant speed-up compared to software implementations, and the training time of EA can be equal to, or less than conventional optimization techniques. This is a primary motivation of this paper. The combination of EA and FPGA is also seen widely in the field of Evolvable Hardware, where slightly different motivations are observed. EHW is often concerned with hardware design itself, and therefore EA are usually applied to more primitive hardware building blocks [1]. In practice, this distinction is often blurred. In the first case, accelerating software EA experiments often requires algorithmic trade-offs in order to implement chromosomes efficiently on FPGAs. In the second case, knowledge of high-level algorithms is often required to apply EHW design principles to practical problems.

This paper uses the combination of EA and FPGA to find hardware efficient solutions to pattern recognition problems. In this paper, the data sets considered are remotely sensed, multi-spectral imagery and the problem is referred to as Automatic Feature Extraction (AFE). AFE attempts to find algorithms that will consistently separate a feature of interest from the background in the presence of noise and uncertain conditions. The design space for hardware efficient AFE comes from two sources: algorithmic components of conventional AFE solutions and hardware resources available in FPGA devices. Network architectures have several properties that lead to efficient hardware implementation. These include:

- Inherent Parallel Processing: The final output of a network is a result of partial calculations performed by each node.
- Simple Processing Elements: Each node of the network need only be capable of solving part of a particular problem and therefore are relatively simple.
- Modular: Nodes are usually homogeneous across the network leading to simple large-scale designs.

For these reasons, networks appear to be a good starting point from which to develop hardware efficient AFE algorithms. This is not a new thing, and is partly why Neural Networks have received considerable attention for solving the classification aspect of AFE problems [2]. However, applying classification directly often leads to poor performance on out-of-sample data and therefore preprocessing and feature extraction are usually required. A good set of features will make classification easier and hopefully lead to good generalization.

One approach to this problem is the co-optimization of feature extraction and classification components. This means feature selection can be directed towards easily classifiable subsets, while simultaneously leading to simpler classifiers. An example of this approach was presented in [3] where a morphological shared weight neural network was used for an automatic target recognition problem. A problem with the co-optimization

0-7695-1180-5/01 $10.00 © 2001 IEEE

approach is that learning algorithms become complex. There are a large number of potentially useful transformations that could be used for feature extraction and optimization soon becomes intractable. This problem can be avoided by using EA.

In Section 2, an FPGA fitness evaluator is presented. It is implemented on the Firebird Custom Computer (CC) by Annapolis Microsystems [4]. This CC is based around a Virtex 2000E FPGA from Xilinx. It has four 64-bit local memories and a 5th 32-bit local memory and communicates with a host computer through a 64-bit PCI bus. The Firebird CC is used throughout the paper to accelerate evolution of network architectures.

Preliminary work is reported in Section 3 where a traditional two-layer Neural Network is compared to a more recently introduced Morphological Network [5] of comparable size. These networks are applied to a range of feature identification problems in multi-spectral imagery in Section 4. Section 5 builds on Section 3 and presents a novel network node that combines spectral classification techniques with spatial enhancement and feature extraction algorithms, in a self contained, modular design. This means *hybrid* feature extraction/ classification architectures that are scalable, inherently parallel and easily implemented. Section 6, makes an assessment of the approach by evolving a 3-layer multi-spectral network of these nodes known as POOKA.

2. Network Fitness Evaluation using CC

Figure 1 illustrates the major components for the CC fitness evaluator. The host/CC communication for a pipelined fitness evaluator is very efficient. Large volume training data is loaded to the CC local memory once, at the start of the run. During evolution, the host only writes to and reads on-chip registers. Large volume result data is retrieved once, at the end of the run.

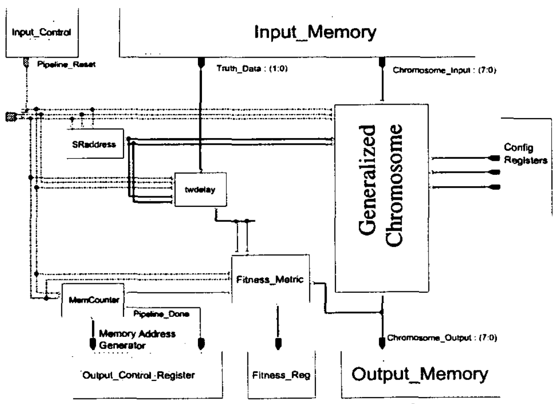


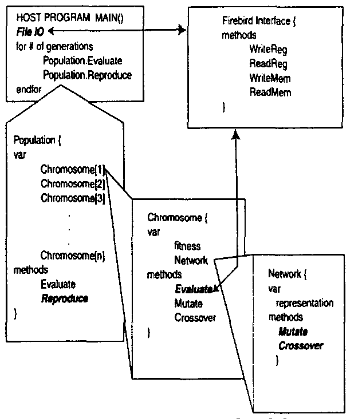
Figure 1: Pipeline Fitness Evaluator

The *Generalized Chromosome* implements the search space. It is configured with the on-chip *Config Registers*. By writing to these registers, a particular chromosome is configured which can then be evaluated. The *Generalized Chromosome* receives training data from one memory, performs the particular processing dictated by the configuration registers and then outputs the result to a second memory.

At the same time the truth data, also loaded to local memory, is passed to a delay unit. This is labeled *twdelay* in Figure 1 and implements latency equal to the *Generalized Chromosome* Pipeline. The latency-adjusted truth is then compared to the chromosome output in the *Fitness-Metric* unit. Input to the *Generalized Chromosome* is assumed to be signed 8-bit integers in the range {-127:127}. The fitness metric applies a threshold at 0. A binary metric can then be used which is a hamming distance weighted by the number of training points in the True and False classes.

$$Fitness = \left({}^{T_C}\!/_{T_T}\right) * 500 + \left({}^{F_C}\!/_{F_T}\right) * 500 \quad \textbf{(1)}$$

T_C is the number of true pixels correctly classified by the network and T_T is the total number of true pixels in the training set. Similarly, F_C is the total number of false pixels correctly classified and F_T is the total number of false pixels in the training set. A perfect classification will result in a score of 1000. Since finding all feature pixels is equivalent to finding all non-feature pixels, two fitness scores are calculated. The host program chooses the larger fitness value and assigns it to the chromosome. The Hamming metric is suitable only for two-class classification problems and only classification error is considered. No measure is made of the certainty in decision such as a distance from the decision boundary. The benefit of the weighted hamming metric is its simplicity of implementation in CC.



```
HOST PROGRAM MAIN()
File IO
for # of generations
        Population.Evaluate
        Population.Reproduce
endfor
```

```
Firebird Interface {
methods
        WriteReg
        ReadReg
        WriteMem
        ReadMem
}
```

```
Population {
var
        Chromosome[1]
        Chromosome[2]
        Chromosome[3]
        .
        .
        .
        Chromosome[n]
methods
        Evaluate
        Reproduce
}
```

```
Chromosome {
var
        fitness
        Network
methods
        Evaluate
        Mutate
        Crossover
}
```

```
Network {
var
  representation
methods
  Mutate
  Crossover
  }
```

Figure 2: Host Program Architecture

The overview of the host program is illustrated in Figure 2. This shows where the major components (bold) are implemented. The *Chromosome* object interfaces the software chromosome, encoded in the *Network* object, with the Firebird. For each chromosome, it sets the *Pipeline-Reset* and writes the configuration registers according to the representation stored in the Network object. It then clears the *Pipeline-Reset* and waits for the *Pipeline-Done* signal. Once received, the chromosome object retrieves the fitness score.

3. Morphological and Linear Perceptrons

Neural Networks have been implemented on CC by several researchers to accelerate both training and application. A fundamental operation in neural networks is multiplication. This can be expensive to implement on Field Programmable Gate Arrays (FPGA) as the number of nodes and connectivity within the network grows. Several techniques have been used to reduce this problem: implementation of partially connected neural networks [6], and time multiplexing of network nodes using run time reconfiguration [7].

Morphological Networks have much in common with Neural Networks but represent a fundamentally different approach. They have been shown to have equivalent classification power to neural networks [8] and can be implemented on FPGAs much more efficiently than neural networks. Traditional neural networks, using linear perceptrons, multiply the inputs by weights, and then sum the result. This linear operation is then followed by a non-linear thresholding operation to produce the perceptron output. This is illustrated in Figure 3a. In the morphological case, the operations of multiplication and addition are replaced by addition and maximum respectively. The morphological perceptron is illustrated in Figure 3b.

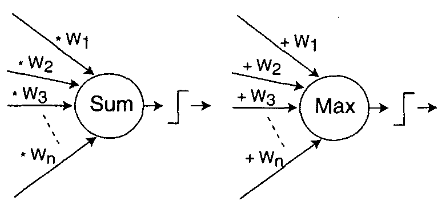


Figure 3: a) Linear and b) Morphological Perceptrons

The definition of the morphological perceptron comes from work presented in [5]. A multiplicative weight of ±1 is also associated with each input, which is described as an excitory/inhibitory weight. Several other researchers have suggested morphological networks, but in other forms [9], [10]. Both morphological and neural networks were implemented on the Firebird CC. Each perceptron had 12 inputs. Two-layer networks were built by combining four perceptrons for both the morphological and linear case. The binary outputs from both morphological and linear perceptrons were combined with a logical AND in the output node. This represents a 12-input, 4 hidden-layer, 1-output node network. Table 1 summarizes the resource requirements for the morphological and neural network implementations. There is a resource overhead when using the Firebird CC for memory, clock and PCI bus interface circuits. An 'empty' design was therefore also implemented so that the relative cost of the network architectures would be clear.

Table 1: Resource Comparison

Design	Virtex Resources (slices)	Total (%)	Design (%)
Firebird Infrastructure	1935	10	0
Morphological Network	2432	12	2
Neural Network	3470	18	8

The neural network is approximately 4 times larger than the morphological design. A timing constraint of 66MHz was easily met by the morphological design with a pipeline latency of 2 clock cycles. The neural network design required 3 pipeline stages and several iterations of Place and Route were required. This suggests relative resource requirements may be greater for larger neural network implementations.

4. Application to Multi-Spectral Data Sets

In multi-spectral image processing, a D element vector in spectral space characterizes each pixel, where D is the number of spectral channels in the image. In most traditional approaches to multi-spectral AFE, classification is applied directly to this D-dimensional space [11]. Both networks were applied in this way to a 10-band multi-spectral data set. The training images that were used are depicted in Figure 4. The truth data, which defines the feature of interest, is depicted in the bright overlay. Non-feature is also specified, to allow some pixels to be don't care which do not contribute to fitness. The training data for non-feature is not shown.

The problem set was designed to span a range of difficulties. In Figure 4a, the feature of interest is water. This is the easiest problem of the three since water has a unique spectral signature. The second problem is to identify the golf courses. It is believed that this problem is of moderate difficulty but should have distinguishable spectral properties. The third training image specifies urban or 'built-up' areas as the feature of interest. Urban areas can include a wide variety of materials and therefore spectral signatures. It is believed this is the hardest problem to be solved with spectral information alone. Truth data was also specified for test images for the golf course and urban area problems so that a score could be obtained, and performance quantified.

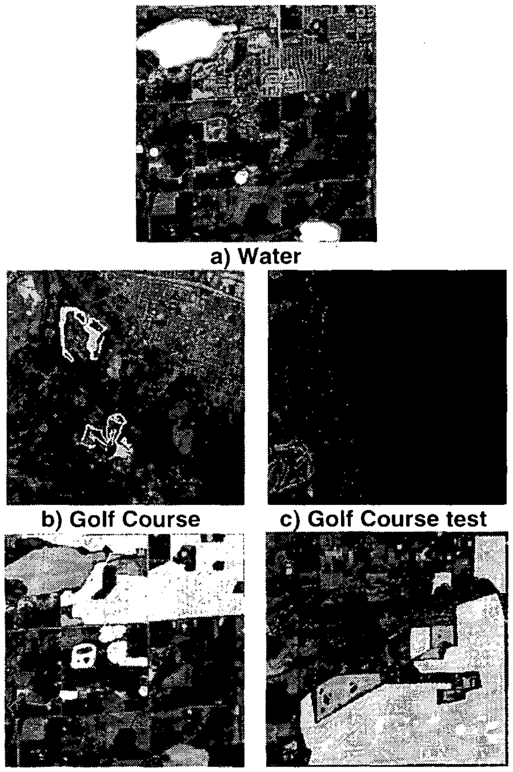


Figure 4: Multi-Spectral Problem Set

Both morphological and neural networks were evolved for 5 independent runs for each training image. The chromosome for the Neural Network is made up of multiplicative weights in the range {-7,7}. For the Morphological Network they are additive weights range {-128, 128: powers of 2} and the inhibitory/excitory weight {-1 or 1}. The search space size for the two networks is therefore approximately the same. A generational Genetic Algorithm with elitism was used [12].

For each run, a population of 200 networks was evolved for 500 generations. The results are summarized in Table 2 where a perfect classification will result in a score of 1000. For comparison, a software experiment was also implemented and a back-propagation learning algorithm was used. Multiple iterations of back-propagation were used, but were often caught local optima. There are many advanced learning algorithms that may find global optima more consistently, but this is also true for the EA. The best networks found in the 5 runs were then applied to the test problems, and results are also reported in Table 2.

Table 2: Summary of Results

Test Problem	**Morph. Network**		**Neural Network**		**Neural Back Prop.**
Training	*Mean*	*SD*	*Mean*	*SD*	
Water	999	0.1	999	0.08	999
Golf	954	3.73	962	1.65	937
Urban	759	9.26	788	7.69	756
Testing	*Best applied*		*Best applied*		
Golf	579		909		768
Urban	660		737		683

The Neural Network architecture outperformed the Morphological Network architecture in all problems. All architectures found the problems progressively more difficult as expected. The following conclusions are made:

1. If a network is to be implemented in hardware, evolutionary search is a well-motivated learning algorithm.
2. The Virtex FPGA can efficiently implement simple multi-bit arithmetic, multiplexing and small fixed-point multipliers.
3. Both spectral and spatial information are important in many feature identification problems of interest.

5. A Multi-Spectral Processing Node

The flexibility of EA suggests development of network architectures more suitable for multi-spectral image processing. Modern multi-spectral sensors are now being produced with high spatial resolution. To exploit these advances in sensor technology feature identification algorithms must utilize both spectral and spatial information. Figure 5 illustrates the spatial-spectral processing node.

The node has four inputs (four multi-spectral bands if applied directly to the data) and one output. The four inputs are first combined with a spectral processor that produces one output. This output is then input to a spatial processor. The two additional *Precision* components are used for controlling bit widths and handling precision.

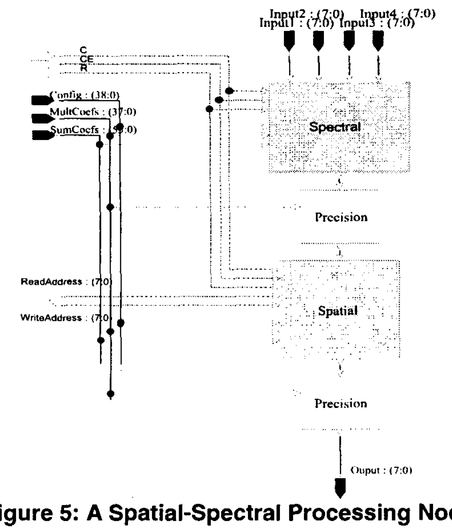


Figure 5: A Spatial-Spectral Processing Node

5.1 The spectral processor

The spectral processor can be considered a *hybrid* network node that incorporates both linear and morphological network functionality. Two coefficients are associated with each image plane and are applied according to Equation 2. The sum coefficients have a range between –127 and 127. The Multiplicative coefficients may assume values between –7 and 7.

$$Output = (Input + Sum_{Coef}) * Mult_{Coef} \qquad \textbf{(2)}$$

Both images then pass to the *Arith-Morph-Mux* (AMM) unit that is illustrated in Figure 6. This processing block incorporates the fundamental flexibility of both spectral and spatial processor. The block is based around a programmable add/subtract unit. With the addition of control logic and multiplexers, the block can be configured to perform a number of functions of two inputs. A particular function is configured by setting 3 control lines *Mux, Func* and *Morph.* The corresponding functions are described in Table 3.

The spectral processor chromosome has four sets of coefficients and configuration bits for 3 Arith-Morph-Mux units (a two-layer binary tree).

5.2 The spatial processor

The Spatial Processor is applied to a single input image and implements functions of a 5 by 5 neighborhood. Figure 7 depicts the 5 by 5 register array associated with the operator. The input image is supplied to the processor through the row0 input. Four row outputs (on the right of the image) input to image-width row buffers whose outputs supply row1 through row4 inputs. Each register has an associated output so that the neighborhood function can be applied.

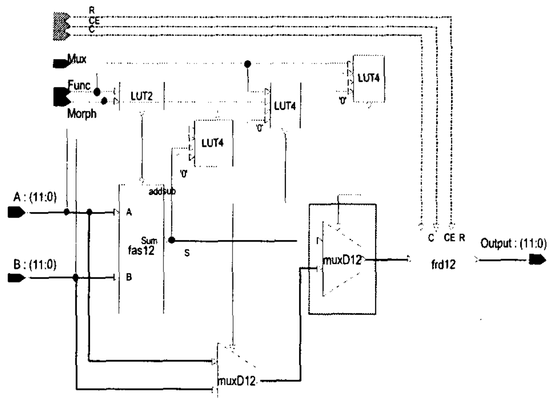


Figure 6: The Arith-Morph-Mux (AMM) unit

Table 3: Functions of the AMM unit

Control Bits (AMM)			Function Applied to pixels p_1 and p_2
Mux	Func	Morph	
0	0	0	Average $(p_1 + p_2)/2$
0	0	1	Difference $(p_1 - p_2)/2$
0	1	0	Maximum $\vee \{p_1, p_2\}$
0	1	1	Minimum $\wedge \{p_1, p_2\}$
1	*	0	Select p_1
1	*	1	Select p_2

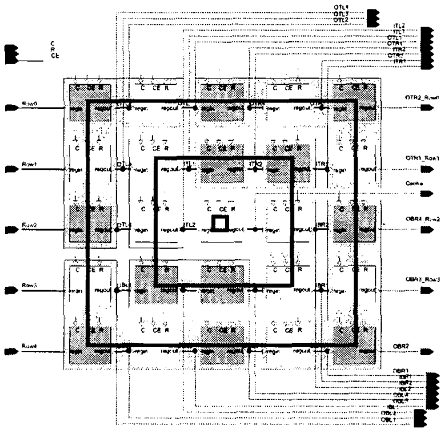


Figure 7: 5by5 Spatial Kernel

The spatial operator uses a particular order to combine the 25 input pixels into 1 output pixel. It is implemented with a binary tree made from *Arith-Morph-Mux* (AMM) units and *Arith-Morph-Abs* (AMA) units. The AMM units were discussed with respect to the Spectral Processor. The AMA unit replaces the multiplexing functionality with an absolute value operation. The modified configuration bits are summarized in Table 4.

Table 4: Configuration of AMA Unit

Control Bits (AMA)			Function
Mux	Func	Morph	Applied to pixels p_1 and p_2
1	*	0	$\mid (p_1 + p_2)/2 \mid$
1	*	1	$\mid (p_1 - p_2)/2 \mid$

Image-processing algorithms motivate the order of combination in the binary tree. At the top level, the 25 inputs are first combined into 3 *rings*. These are superimposed on Figure 5. The 5by5 ring has 16 inputs and the 3by3 ring has 8. The 3rd ring is the center pixel. These 3 values are combined with a spectral processor of 3 inputs. There are therefore both multiplicative and additive coefficients associated with each ring.

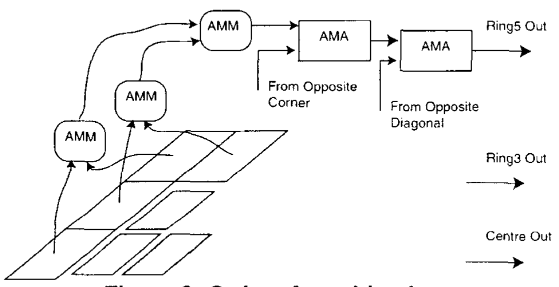


Figure 8: Order of combination (top-left corner of 5by5 mask)

The order in which pixels are combined in the ring is important. The order for the 5by5 case is illustrated in Figure 8. First, the corners of each ring are found. In the 5by5 case, a 4-input network of AMM units is used. This can be configured to find the average, maximum or minimum of the corner pixels (or any subset of). The corner averages can be used to estimate a gradient by combining opposite corners with the AMA unit. In this case, an absolute value of the difference represents the magnitude of an edge response [13]. Once opposite corners have been combined, the two diagonals that result, are combined with another AMA unit. This is most clearly seen in Figure 8.

Each ring can return an average, maximum, minimum or edge response. By associating weights with these rings, a *hybrid* linear/non-linear spatial filter is implemented. By setting weights appropriately, Gaussian smoothing and simple combinations of Gaussian functions can be implemented [14]. In this respect, the architecture is similar to that found in Convolutional Neural Networks [15]. However, in addition, the morphological aspect of the spatial filter means a rich variety of non-linear spatial filters can also be implemented. Examples of these include erosion, dilation and morphological range operators [16]. Aspects of the spatial processor worth particular note:

1. The combination of edge responses and smoothing is optimized by the EA. This is a powerful measure of texture. Convolution kernels by Laws [17] and their modifications in [18] are excellent examples of this type of linear filter.
2. The combination of linear and non-linear components is also optimized by the EA. For example, linear combinations of erosions and dilations. This is similar to hybrid L-filters [19] and pseudo-granulometries [20].

To encourage rotationally invariant operators, and to reduce the size of the search space, only one quarter of the tree is configured. The configuration for the top left quadrant of the tree is used in the other three quadrants. This is a common way of enforcing rotationally invariant structuring elements when optimizing morphological filters. Figure 9 illustrates the technique. Only the top-left portion of the neighborhood with gray background is configured. This configuration is then rotated through the four quadrants. In this example, a particular configuration produces a filter that depends only on pixels that are crossed. In terms of morphology this is known as a structuring element and the result can be seen on the right of Figure 9.

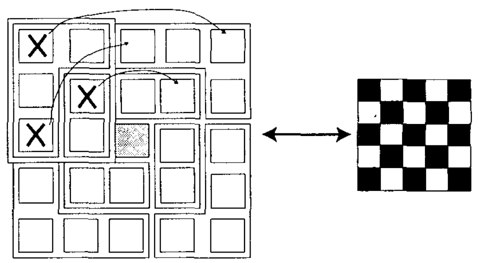


Figure 9: Shared Configuration

The spatial processor chromosome has three sets of coefficients associated with the 3 rings. With shared configuration, the AMM/AMA network requires 10 sets of the *mux, func* and *morph* configuration bits.

5.3 Band selection and precision

Input to the node is assumed to be 8 bit 2's complement values. Local memory resources of the Firebird CC dictated an upper limit of 12 channels. Each input to the node can receive input from any of the 12 channels. To accommodate a variable number of bands, tri-state bus resources were used to implement large multiplexers.

Input data is first scaled to the range {−127:127}. The multiplicative weights, used to combine inputs in the Spectral Processor, and the center, 3by3 and 5by5 rings in the Spatial Processors produce 12 bit signed outputs at full precision. It is desirable to maintain a consistent bit-width between input and output so that nodes can be easily cascaded to form networks. In this case, the EA is used to select which 8-bits should be used in the 12-bit output. This effectively scales the output, by dividing by powers of 2. Tri-state buses are used to multiplex the twelve bit input data. By setting control lines the bus effectively divides by 1,2,4,8 or 16. Data outside of the range {-127:127} after this scaling will saturate. This is a good example of how EA can be used to find solutions within a design space of finite hardware resources.

5.4 Node representation

The general structure of the node chromosome can be seen in Figure 10. The chromosome is made up of a combination of BITS, found in the AMM and AMA units, and an additional absolute value operation implemented in the precision unit. There are also integers, used in spectral/spatial processor coefficients, and in the precision unit divider.

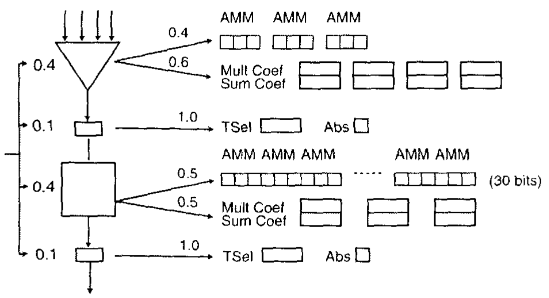


Figure 10: Node representation and mutation tree

Similar to the hardware, the chromosome is stored hierarchically in a number of objects in software. When mutation and crossover are applied to the node, there is a certain probability that they affect particular components. This can be considered a probability tree and Figure 10 shows the values used for mutation. Once mutation reaches a terminal node, there is equal probability of mutation within the subgroup. Crossover points are chosen in a similar way but are not discussed further.

6. POOKA: A Multi-Spectral Network

A 3-layer, 9 node network was implemented. The *Generalized Chromosome* for this network is illustrated in Figure 11. There are 16 inputs to the network at the first layer and therefore a total of 16 multi-spectral channels are chosen from the training data. This results in an additional 16 integers in the network chromosome. More than 1 node can get input from the same image channel. Each node in the second layer receives input from the four outputs of the first layer. The order of inputs for second layer nodes is kept constant for crossover. This means the first input of the 15 node is the same first input for 16, 17 and 18 nodes.

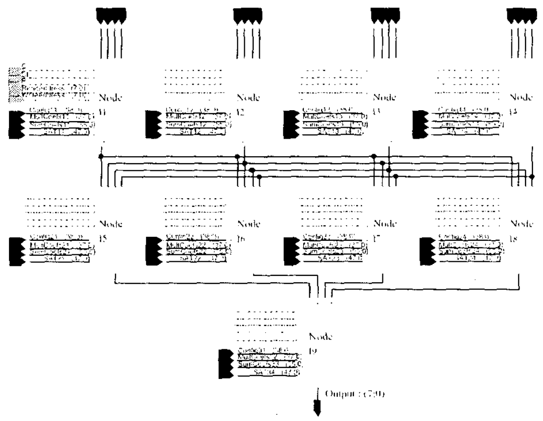


Figure 11: The 3-layer 9-node network

Several extensions to the Fitness Evaluator architecture of Section 2 were required to implement this larger network. These can only be briefly described in this paper. The output from each node is sent to local memory so that the host can retrieve it. Each node is also allocated a separate *twdelay* and *Fitness Metric* unit. This means fitness can be calculated on the output from each node. The training data associated with each node is also flexible. This means a total of 9 target classifications can be supplied to the network, one for each node.

Several extensions of the Host Program were also required. Multiple populations were used, each associated with a network node. Evolution within each population was kept independent to encourage specialization. This also means *incremental learning* techniques [21] can be implemented easily. Incremental evolution of the POOKA network can be considered a 3-stage process:

1. The four 1st layer nodes are evolved in parallel in four different populations. Since a fitness is calculated on the output from each node, these populations can be evolved independently.
2. In the second stage, the best 1st layer nodes in each population are configured and remain fixed. The 4 nodes in the 2nd layer are then evolved independently in 4 populations.
3. In the third stage, the best 2nd layer nodes are also configured. Both 1st and 2nd layer nodes remain fixed and the output node is evolved.

To maximally utilize the fitness evaluator resources, all 9 nodes should be involved in evolution at all times. This is not possible with the *Incremental Learning* approach. It is possible to evolve higher layers while lower-level nodes are evolved. This means the 1st, 2nd and 3rd layers are evolved in Stage 1. Only the 2nd and 3rd are evolved in Stage 2 and just the 3rd layer in Stage 3. This is the approach used in this paper, and is illustrated in Figure 12.

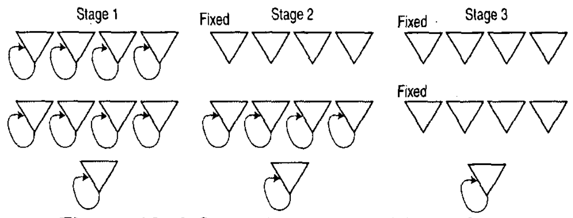


Figure 12: 3-Stage Incremental Learning

After *Incremental Evolution*, a variable number of *Optimization Cycles* are applied. This is most similar to *greedy* strategy suggested in [21] and is used to encourage collaboration between network nodes. This is an important concept in evolutionary neural networks and complex co-evolutionary strategies have been suggested [22]. In POOKA, optimization is a 9-stage process. The best nodes found after incremental evolution are configured. Each node is then taken in turn, and evolved for a number of generations with fitness calculated on the final output from the network (3rd layer output). This is illustrated in Figure 13. The Optimization Cycle is used to promote nodes that may not score well individually, but lead to better scores in the network as a whole.

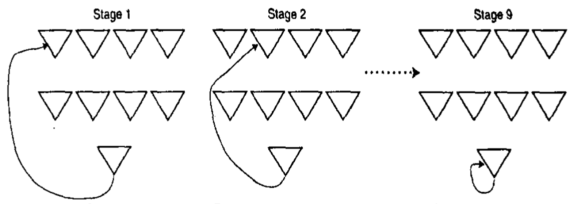


Figure 13: 9-Stage Optimization Cycle

6.1 Resource usage and evaluation of speed-up

The 9-node, 3-layer network was implemented at 50MHz and approximately 64% of the Virtex 2000E FPGA. Pre Place and Route the usage was estimated at 45%. This indicates significant room to optimize the design. All components of the network and fitness evaluator architectures were designed with structural VHDL to which placement constraints can be applied. This effectively allows the design to be manually placed, which can significantly improve density and clock rates. Evaluating speed-up of the architecture compared to software implementations is a difficult problem. Raw processing speed is not the only factor, since quality of the feature extraction algorithm is also important.

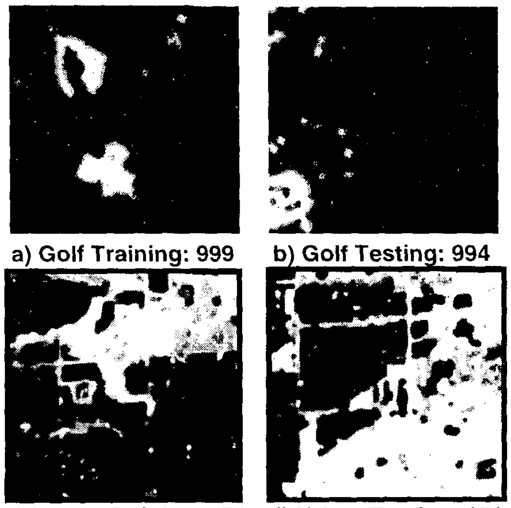
In the case of raw processing speed, one measure of performance is estimated by considering a high-level approximation of network components. In this case, the quality of algorithm is not considered, but rather the execution time of a particular chromosome. For the software, execution time was estimated by implementing a number of optimized image processing operators. For each Spectral Processor in the network, a linear combination was used. For each Spatial Processor, a 5by5 neighborhood average was calculated. The software experiment performed a total of 9 linear combinations of 4 images and 9 5by5 neighborhood averages. The execution times and relative speed-up are summarized in Table 5.

Table 5: Evaluation of Speed-up

Image Size (pixels)	Software Evaluation (Seconds)	RC Evaluation (Seconds)	Speedup
65536	0.18	0.0016	112
131072	0.36	0.0029	124
262144	0.71	0.0055	129
524288	1.39	0.0105	122
1048576	2.75	0.0201	136

6.2 Application to multi-spectral data sets

The network was trained on the 3 multi-spectral problems used in Section 4. The training time for each problem, including reading and writing training data was approximately 54.5 seconds. This included 52 seconds in evolution: 18.7 seconds of incremental development, followed by two optimization cycles of 16 seconds each. Populations of 200 nodes were evolved for approximately 180 generations.



c) Urban Training: 979 d) Urban Testing: 959
Figure 10: Output images from POOKA (fitness out of 1000)

Results for the water identification problem are not shown since the problem was easily solved and a perfect classification was obtained on the training data. Similar performance was observed for this problem in Section 4. The output images and fitness scores for the golf course and urban area problems are shown in Figure 10. Output images from the training data are shown on the left, and the output found on the test images are shown on the right. Fitness scores are calculated using Equation 1, and a perfect classification results in a score of 1000.

The POOKA network out-performed the morphological and neural networks of Section 4 on all problems. This is not surprising since these networks did not use spatial information. A more comprehensive comparison has been made to advanced spatio-spectral software techniques and results have been promising. This comparison will be presented in future publications.

7. Discussion and Future Directions

Network architectures are naturally suited to implementation on CC. Such architectures are easily scalable and inherently parallel. Evolutionary search is an effective means to optimize network architectures if the training time is reasonable. By implementing networks on CC, evolutionary search is a particularly attractive learning algorithm. The flexibility of evolutionary search means network architectures can be implemented with a particular problem in mind.

A novel network node was suggested for multi-spectral feature identification. It combines both spectral and spatial information using an image processing inspired, *morpho-linear* network. A 3-layer network of these nodes was described and preliminary results reported. Speed-up of two orders of magnitude compared to a software implementation of similar complexity was achieved. The effectiveness of evolutionary search, applied to large networks, can be improved by implementing multiple *Fitness Metric* units. This also has the advantage of allowing different target classifications to be associated with different nodes. Such flexibility may be used to direct evolution for more difficult problems. For example, a beach finder may be formed by combination of other high-level features such as land and water. Multiple *Fitness Metric* units allow some nodes to be trained to find water, and others to find beach, in parallel.

An interesting future direction will be to extend the network architecture presented by introducing *state*. This can be readily implemented with the Fitness Evaluator architecture since the output from each node is stored in the local memory. These output images can be considered the current state of the node. They can be fed back, as input to the node, to implement functions of state. Such architectures have several interpretations. In one sense, the network implements multi-layered gray-scale cellular automata [23], in which the transition rules are an engineered subset drawn from image processing. Another interpretation is a multi-layer cellular neural network [24], or more accurately, a cellular *morpho-linear* network. Fitness evaluation for these types of architectures will require multiple passes of the training data for each chromosome. These architectures are potentially applicable to real-time multi-spectral image processing.

8. References

1. Miller, J.F., D. Job, and V.K. Vassilev, *Principles in the Evolutionary Design of Digital Circuits - Part I.* Genetic Programming and Evolvable Machines, 2000. **1**(1): p. 7-35.
2. Bishop, C.M., *Neural Networks for Pattern Recognition.* 1995, Oxford: Oxford University Press.
3. Won, Y. and P.D. Gader. *Morphological Shared-Weight Neural Network for Pattern Classification and Automatic Target Detection.* in *IEEE International Conferenec on Neural Networks.* 1995.
4. Annapolis, M., *http://www.annapmicro.com/.* 2001.
5. Ritter, G.X. and P. Sussner. *An introduction to morphological neural networks.* in *13th International Conference on Pattern Recognition.* 1996. Vienna, Austria.
6. Chung, Y.Y., et al. *Implementing Neural Network in Custom Computers.* in *IEEE International Conference on Systems, Man and Cybernetics : Conference Theme : Intelligent Systems for Humans in a Cyberworld.* 1998. San Diego, California, USA: IEEE.
7. Eldredge, J.G. and B.L.Hutchings, *Run-Time Reconfiguration: a method for enhancing the functional density of SRAM-based FPGAs.* Journal of VLSI Signal Processing, 1996. **12**(1): p. 67-86.
8. Sussner, P. *Morphological Perceptron Learning.* in *Joint Conference on the Science and Technology of Intelligent Systems.* 1998. Maryland: IEEE.
9. Wilson, S.S. *Morphological Networks.* in *Visual Communications and Image Processing IV.* 1989: SPIE.
10. Yang, P. and P. Maragos, *Min-Max Classifiers: Learnability, Design and Application.* Pattern Recognition, 1995. **28**(6): p. 879-899.
11. Figueiredo, M.A. and C. Gloster. *Implementation of a Probabilistic Neural Network for Multi-spectral Image Classification on an FPGA Based Custom Computing Machine.* in *Vth Brazilian Symposium on Neural Networks.* 1998. Belo Horizonte, Brazil: IEEE Computer Society.
12. Mitchell, M., J.P. Crutchfield, and R. Das. *Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work.* in *First International Conference on Evolutionary Computation and Its Applications (EvCA'96).* 1996. Moscow, Russia.
13. Jain, A.K., *Fundamentals of Digital Image Processing.* Pretice Hall Information and System Sciences Series. 1989, New Jersey: Pretice Hall.
14. Macleod, I.D. and A. Rosenfeld, *The visibility of gratings: Spatial frequency channels or bar detecting units.* Vision Research, 1974. **14**: p. 909-916.
15. LeCun, Y. and B. Boser, *Convolutional networks for images, speech and time series,* in *The Handbook of Brain Science and Neural Networks,* M. Arbib, Editor. 1995, MIT Press: Cambridge, MA. p. 255-258.
16. Woods, R.C.G.a.R.E., *Digital Image Processing.* 1993, Reading, Massachusetts: Addison-Wesley Publishing Company.
17. Laws, K.I. *Texture energy measures.* in *Proceedings of Image Understanding Workshop.* 1979.
18. Pietikainen, M., A. Rosenfeld, and L.S. Davis, *Experiments with Texture Classification Using Averages of Local Pattern Matches.* IEEE Transactions on Systems, Man and Cybernetics, 1983. **SMC-13**(3).
19. Bovik, A.C., T. Huang, and D. Munson, *A generalization of median filtering using linear combinations or order statistics.* IEEE Trans. Acoust., Speech, Signal Processing, 1983. **31**: p. 1342-1350.
20. Aubert, A., D. Jeulin, and R. Hashimoto. *Surface Texture Classification from Morphological Transformations.* in *5th International Symposium on Mathematical Morphology.* 2000. Palo Alto, California: Kluwer Academic Publishers.
21. Potter, M.A. and K.A.D. Jong. *Evolving Neural Networks with Collaborative Species.* in *Proceedings of the 1995 Summer Computer Simulation Conference.* 1995. Ontario, Canada.
22. Moriarty, D.E. and R. Miikkulaiinen, *Forming Neural Networks through Efficient and Adaptive Coevolution.* Evolutionary Computation, 1998. **5**(4).
23. Sahota, P., M.F. Daemi, and D.G. Elliman. *Using Genetically Evolving Multi-Layer Cellular Automata for Image Processing.* in *Third Golden West Ineternational Conference on Intelligent Systems.* 1995. Netherlands: Kluwer Academis Publishers.
24. Chua, L.O., *CNN: A Paradigm for Complexity.* 1998: World Scientific Publishing Company.

Experimental Testing of the Gigabit IPSec-Compliant Implementations of Rijndael and Triple DES Using SLAAC-1V FPGA Accelerator Board

Pawel Chodowiec[1], Kris Gaj[1], Peter Bellows[2], and Brian Schott[2]

[1] Electrical and Computer Engineering, George Mason University, 4400 University Drive, Fairfax, VA 22030
{kgaj, pchodow1}@gmu.edu
[2] University of Southern California - Information Sciences Institute
Arlington, VA 22203
{pbellows, bschott}@east.isi.edu

Abstract. In this paper, we present the results of the first phase of a project aimed at implementing a full suite of IPSec cryptographic transformations in reconfigurable hardware. Full implementations of the new Advanced Encryption Standard, Rijndael, and the older American federal standard, Triple DES, were developed and experimentally tested using the SLAAC-1V FPGA accelerator board, based on Xilinx Virtex 1000 devices. The experimental clock frequencies were equal to 91 MHz for Triple DES, and 52 MHz for Rijndael. This translates to the throughputs of 116 Mbit/s for Triple DES, and 577, 488, and 423 Mbit/s for Rijndael with 128-, 192-, and 256-bit keys respectively. We also demonstrate a capability to enhance our circuit to handle the encryption and decryption throughputs of over 1 Gbit/s regardless of the chosen algorithm. Our estimates show that this gigabit-rate, double-algorithm, encryption/decryption circuit will fit in one Virtex 1000 FPGA taking approximately 80% of the area.

1. Introduction

IPSec is a set of protocols for protecting communication through the Internet at the IP (Internet Protocol) Layer [15, 22]. One of the primary applications of this protocol is an implementation of Virtual Private Networks (VPNs). In IPSec Tunnel Mode, multiple private local area networks are connected through the Internet as shown in Fig. 1a. Since the Internet is an untrustworthy network, a secure tunnel must be created between security gateways (such as firewalls or routers) belonging to private networks involved in the communication. The information passing through the secure tunnel is encrypted and authenticated. Additionally, the original IP header, containing the sender's and receiver's addresses is also encrypted, and replaced by a new header including only information about the security gateway addresses. This way a limited resistance against the traffic control analysis is accomplished. A second use of IPSec is client-to-server or peer-to-peer encryption and authentication (see Fig. 1b). In IPSec Transport Mode , many independent pair-wise encryption sessions may exist

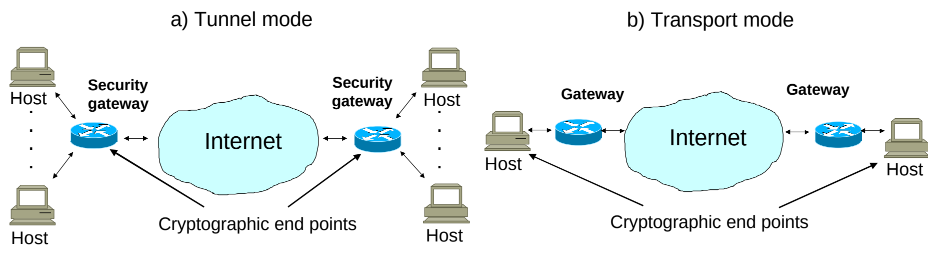


Fig. 1. IPSec Tunnel and Transport Modes

simultaneously. *The large number of connections and high bandwidth supported by a single security gateway or server suggests the use of hardware accelerators for implementing cryptographic transformations.*

The suite of cryptographic algorithms used for encryption and authentication in IPSec is constantly evolving. In the case of encryption, current implementations of IPSec are required to support DES, and have the option of supporting Triple DES, RC5, IDEA, Blowfish, and CAST-128. Since DES has been shown to be vulnerable to an exhaustive key-search attack using the computational resources of a single corporation [2], the current implementations of IPSec typically support Triple DES. In 1997, the National Institute of Standards and Technology (NIST) initiated an effort towards developing a new encryption standard, called AES (Advanced Encryption Standard) [1]. The development of the new standard was organized in the form of a contest coordinated by NIST. In October 2000, Rijndael was announced as the winner of the contest and a future Advanced Encryption Standard. In November 2000, a first Internet-draft was issued, proposing including AES-Rijndael as a required encryption algorithm in IPSec, with the remaining AES contest finalists as optional algorithms to be used in selected applications [11].

An encryption algorithm is not the only part of IPSec that is currently being extended and modified. Other modifications currently being considered include different modes of operation for encryption algorithms [18], hash functions used by authentication algorithms [21], type and parameters of public key cryptosystems used by a key management protocol, etc. *The fast and hard to predict evolution of IPSec algorithms leads naturally to prototype and commercial implementations based on reconfigurable hardware*.

An FPGA implementation can be easily upgraded to incorporate any protocol changes without the need for expensive and time-consuming physical design, fabrication, and testing required in case of ASICs. Additional capabilities appear when an FPGA accelerator supports a real-time partial reconfiguration. In this case, the accelerator can reconfigure itself on the fly to adapt to

- traffic conditions (*e.g.*, by changing the number of packet streams processed simultaneously),
- phase of the protocol (*e.g.*, by using the same FPGA with time sharing for implementing key exchange, encryption, and authentication),
- various key sizes and parameter values (*e.g.*, by adjusting the circuit architecture to different key sizes and different values of system parameters).

Additionally, several optional IPSec-compliant encryption, authentication, and key exchange algorithms can be implemented, and their bitstreams stored in the cache memory on the FPGA board. Algorithm agility accomplished this way can substantially increase the system interoperability.

In this paper, we present the results of the first phase of our project aimed at implementing a full suite of IPSec cryptographic transformations using SLAAC-1V FPGA board. In this phase, two encryption algorithms AES-Rijndael and Triple DES were implemented and experimentally tested in our environment.

2. FPGA Board

The SLAAC-1V PCI board is an FPGA-based computation accelerator developed under a DARPA-funded project called Systems-Level Applications of Adaptive Computing (SLAAC). This project, led by USC Information Sciences Institute (ISI), investigated the use of adaptive computing platforms for open, scalable, heterogeneous cluster-based computing on high-speed networks. Under the SLAAC project, ISI developed several FPGA-based computing platforms and a high-level distributed programming model for FPGA-accelerated cluster computing [16]. About a dozen universities and research labs are using SLAAC-1V for a variety of signal and image processing applications.

The SLAAC-1V board architecture is based on three user-programmable Xilinx Virtex XCV-1000-6 FPGA devices. Each of these devices is composed of 12,288 basic logic cells referred to as CLB (Configurable Logic Block) slices, and includes 32 4-kbit blocks of synchronous, dual-ported RAM. All devices can achieve synchronous system clock rates up to 200 MHz, including input/output interface.

The logical architecture of SLAAC-1V is shown in Fig. 2. The three Virtex 1000 FPGAs (denoted as X0, X1, and X2) are the primary processing elements. They are connected by a 72-bit “ring” path as well as a 72-bit shared bus. The width of both buses supports an 8-bit control tag associated with each 64-bit data word. The direction of each line of both buses can be controlled independently. The processing elements are connected to ten 256K x 36-bit SRAMs (Static Random Access Memories) located on mezzanine cards. The FPGAs X1 and X2 are each connected to four SRAMs, while X0 is connected to two. The memory cards have passive bus switches that allow the host to directly access all memories through X0.

About 20% of the resources in the X0 FPGA are devoted to the PCI interface and board control module. The remaining logic of this device (as well as the entire X1 and X2 FPGAs) can be used by the application developer. The 32/33 control module release uses the Xilinx 32-bit 33MHz PCI core. The control module provides high-speed DMA (Direct Memory Access), data buffering, clock control (including single-stepping and frequency synthesis from 1 to 200 MHz), user-programmable interrupts, etc. The current 32/33 control module has obtained DMA transfer rates of over 1 Gbit/s (125 MB/s) from the host memory, very near the PCI theoretical maximum. The bandwidth for SLAAC-1V using the 64-bit 66MHz PCI controller (using the Xilinx 64-bit 66MHz core) has been measured at 2.2 Gbit/s. The user’s design located in X0 is connected to the PCI core via two 256-deep, 64-bit wide FIFOs. The DMA

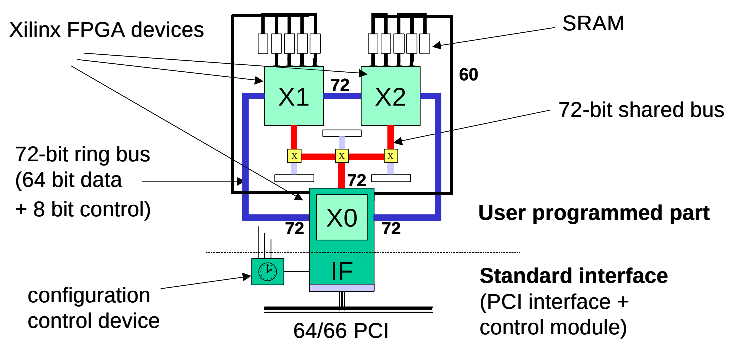


Fig. 2. SLAAC-1V Architecture

controller located in the interface part of X0 can transfer data to or from these FIFOs as well as to provide fast communication between the host and the board SRAMs. The DMA controller load balances input and output FIFOs and can process large memory buffers without host processor interaction. Current interface development includes managing memory buffer rings on the FPGA to minimize host interrupts on small buffers.

SLAAC-1V supports partial reconfiguration, in which part of an FPGA is reconfigured while the rest of the FPGA remains active and continues to compute. A small dedicated Virtex 100 configuration control device is used to configure all FPGAs and manages 6 MB of flash / SRAM as a configuration "cache".

The work discussed in this paper was done in collaboration with the ISI Gigabit-Rate IPSec (GRIP) project, which is funded in the DARPA Next Generation Internet (NGI) program. The GRIP team has constructed a gigabit Ethernet daughter card which connects to SLAAC-1V in place of the crossbar connection of the X0 chip. To the host, the SLAAC-1V / GRIP system appears to be a gigabit Ethernet card with optional acceleration features. The GRIP team is currently customizing the TCP/IP stack for the Linux operating system to take advantage of the hardware acceleration in order to deliver fully-secure, fully-authenticated gigabit-rate traffic to the desktop.

3. Implementation of Rijndael

Rijndael is a symmetric key block cipher with a variable key size and a variable input/output block size. Our implementation supports all three key sizes required by the draft version of the AES standard, 128, 192, and 256 bits. Our key scheduling unit is referred to as 3-in-1, which means that it can process all three key sizes. Switching from one key size to the other is instantaneous, and is triggered by the appropriate control signals. Our implementation is limited to the block size of 128-bits, which is the *only* block size required by Advanced Encryption Standard. Implementing other block sizes, specified in the original, non-standardized description of Rijndael is not justified from the economical point of view, as it would substantially increase circuit area and cost without any substantial gain in the cipher security.

Rijndael is a substitution-linear transformation cipher based on S-boxes and operations in the Galois Fields. Below we describe the way of implementing all component operations of Rijndael, and then present how these basic operations are combined together to form the entire encryption/decryption unit.

3.1 Component Operations

Implementation of the encryption round of Rijndael requires realization of four component operations: ByteSub, ShiftRow, MixColumn, and AddRoundKey. Implementation of the decryption round of Rijndael requires four inverse operations InvByteSub, InvShiftRow, InvMixColumn, and AddRoundKey.

ByteSub is composed of sixteen identical 8x8 S-boxes working in parallel. ***InvByteSub*** is composed of the same number of 8x8-bit inverse S-boxes. Each of these S-boxes can be implemented independently using a 256 x 8-bit look-up table.

A Virtex XCV-1000 device contains 32 4-kbit Block Select RAMs. Each of these memory blocks is a synchronous, dual-ported RAM with the data port width configurable to an arbitrary power of two in the range from 1 to 16. Each memory block can be used to realize two table look-ups per clock cycle, one for each data port.

In particular, each 4-kbit Block Select RAM can be configured as a 512 x 8-bit dual-port memory. If encryption or decryption are implemented separately, only the first 256 bytes of each memory block are utilized as a look-up table. If encryption and decryption are implemented together within the same FPGA, both uninverted and inverted 256 byte look-up tables are placed within one memory block. In each case, 16 data bits are processed by one memory block, which means that a total of 8 memory blocks are needed to process the entire 128-bit input.

ShiftRow and ***InvShiftRow*** change the order of bytes within a 16-byte (128-bit) word. Both transformations involve only changing the order of signals, and therefore they can be implemented using routing only, and do not require any logic resources, such as CLBs or dedicated RAM.

The ***MixColumn*** transformation can be expressed as a matrix multiplication in the Galois Field GF(2^8):

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} \tag{1}$$

Each symbol in this equation (such as A_i, B_i, '03') represents an 8-bit element of the Galois Field. Each of these elements can be treated as a polynomial of degree seven or less, with coefficients in {0,1} determined by the respective bits of the GF(2^8) element. For example, '03' is equivalent to '0000 0011' in binary, and to

$$C(x) = 0\cdot x^7 + 0\cdot x^6 + 0\cdot x^5 + 0\cdot x^4 + 0\cdot x^3 + 0\cdot x^2 + 1\cdot x + 1\cdot 1 = x + 1 \tag{2}$$

in the polynomial basis representation.

The multiplication of elements of GF(2^8) is accomplished by multiplying the corresponding polynomials modulo a fixed irreducible polynomial

$$m(x) = x^8 + x^4 + x^3 + x + 1. \tag{3}$$

For example, multiplying a variable element $A=a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0$ by a constant element '03' is equivalent to computing

$$\begin{aligned} B(x) &= b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0 = \\ &=(a_7 x^7 + a_6 x^6 + a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0) \cdot (x+1) \\ &\bmod (x^8 + x^4 + x^3 + x + 1). \end{aligned} \tag{4}$$

After several simple transformations

$$\begin{aligned} B(x) &= (a_7 + a_6)\ x^7 + (a_6 + a_5)\ x^6 + (a_5 + a_4)\ x^5 + (a_4 + a_3 + a_7)\ x^4 + (a_3 + a_2 + a_7)\ x^3 + \\ &+ (a_2 + a_1)\ x^2 + (a_1 + a_0 + a_7)\ x + (a_0 + a_7), \end{aligned} \tag{5}$$

where '+' represents an addition modulo 2, *i.e.* an XOR operation.

Each bit of a product B, can be represented as an XOR function of at most three variable input bits, *e.g.*, $b_7 = (a_7 + a_6)$, $b_4 = (a_4 + a_3 + a_7)$, etc.

Each byte of the result of a matrix multiplication (1) is an XOR of four bytes representing the Galois Field product of a byte A_0, A_1, A_2, or A_3 by a respective constant. As a result, the entire MixColumn transformation can be performed using two layers of XOR gates, with up to 3-input gates in the first layer, and 4-input gates in the second layer. In Virtex FPGAs, each of these XOR operations requires only one lookup table (*i.e.*, a half of a CLB slice).

The ***InvMixColumn*** transformation can be expressed as a following matrix multiplication in $GF(2^8)$.

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{6}$$

The primary differences, compared to MixColumn, are the larger hexadecimal values of the matrix coefficients. Multiplication by these constant elements of the Galois Field leads to the more complex dependence between the bits of a variable input and the bits of a respective product. For example, the multiplication A='0E' · B, leads to the following dependence between the bits of A and B:

$$a_7 = b_7 + b_6 + b_5 + b_4 \tag{7}$$
$$a_6 = b_6 + b_5 + b_4 + b_3 + b_7 \tag{8}$$
$$a_5 = b_5 + b_4 + b_3 + b_2 + b_6 \tag{9}$$
$$a_4 = b_4 + b_3 + b_2 + b_1 + b_5 \tag{10}$$
$$a_3 = b_3 + b_2 + b_1 + b_0 + b_6 + b_5 \tag{11}$$
$$a_2 = b_2 + b_1 + b_0 + b_6 \tag{12}$$
$$a_1 = b_1 + b_0 + b_5 \tag{13}$$
$$a_0 = b_0 + b_7 + b_6 + b_5. \tag{14}$$

The entire InvMixColumn transformation can be performed using two layers of XOR gates, with up to 6-input gates in the first layer, and 4-input gates in the second layer. Because of the use of gates with the larger number of inputs, the InvMixColumn transformation has a longer critical path compared to the MixColumn transformation, and the entire decryption is more time consuming than encryption.

AddRoundKey is a bitwise XOR of two 128-bit words and can be implemented using one layer of 128 look-up tables, which translates to 64 CLB slices. Assuming that one operand of the bitwise XOR is fixed, this operation is an inverse of itself, so no special transformation is required for decryption.

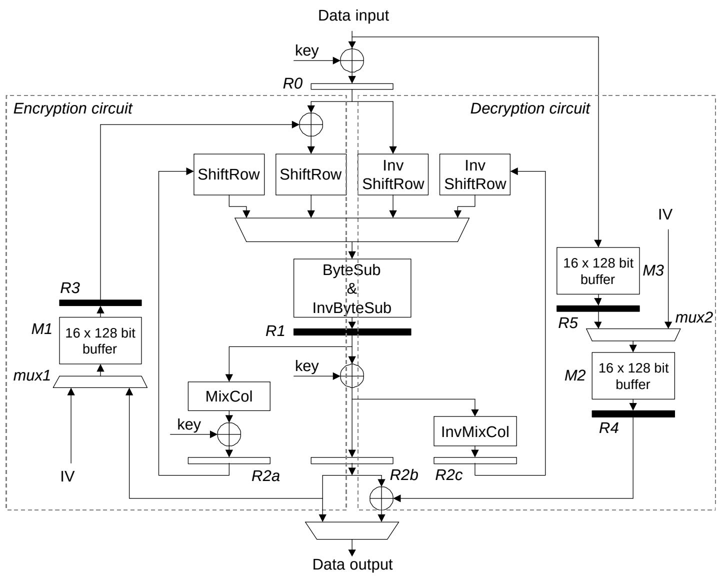


Fig. 3. The general architecture of Rijndael

3.2 General Architecture of the Encryption/Decryption Unit

The block diagrams of the encryption/decryption unit in the basic iterative architecture and in the extended pipelined architecture are shown in Fig. 3. Only registers R1, R3, R4, and R5 (shaded rectangles in Fig. 3) are present in the basic iterative architecture. The remaining registers (transparent rectangles in Fig. 3) have been added in the extended architecture based on the concept of inner-round pipelining.

The register R1 is a part of Block SelectRAM, the synchronous dedicated memory, used to implement ByteSub and InvByteSub transformations, so it was chosen as a basic register in the basic iterative architecture. In this architecture, 11, 13, and 15 clock cycles are required in order to process one block of data for 128-, 192-, and 256-bit keys respectively. The critical path is located in the *decryption circuit*, and includes AddRoundKey (an xor operation), InvMixColumn, InvShiftRow, multiplexer, and InvByteSub (memory read). It is important to note that our *decryption circuit* has a structure (order of operations) similar to the *encryption circuit*, but still does not require any additional processing of round keys (unlike the architecture suggested in [5] and adopted in [8, 9, 10]).

Introducing pipeline registers R2a-c and R0 allows the circuit to process two independent streams of data at the same time. Our architecture assumes the use of the Cipher Block Chaining (CBC) mode for processing long streams of data. The CBC mode is the only mode required by the current specification of IPSec to be used with DES and all optional IPSec encryption algorithms. It is also most likely to be the first mode recommended for use together with AES. The encryption and decryption

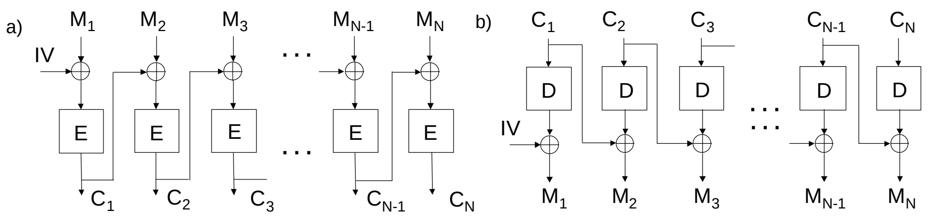


Fig. 4. Cipher Block Chaining Mode a) encryption, b) decryption

in the CBC mode are shown in Fig. 4. An initialization vector IV is different for each packet and is transmitted in clear as a part of the packet header. The CBC mode allows concurrent encryption of blocks belonging to different packets, but not to the same packet. This limitation comes from the fact that the encryption of any block of data cannot begin before the ciphertext of the previous block becomes available (see Fig. 4a). The same limitation does not apply to decryption, where all blocks can be processed in parallel.

In our implementation, the memory buffers M1, M2, and M3 are used to store the last (*i.e.*, the most recently processed) ciphertext blocks for up to 16 independent streams of data. Before the processing of the given stream begins, the corresponding memory location is set to the initialization vector used during the encryption or decryption of the first block of data.

Our architecture allows the simultaneous encryption of two blocks belonging to two different packets, and the simultaneous decryption of two blocks belonging to the same packet or two different packets.

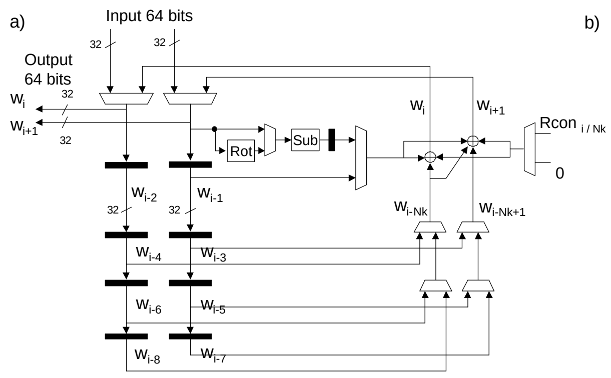
The new secret-key block cipher modes, currently under investigation by NIST, are likely to allow unlimited parallel encryption and decryption of blocks belonging to the same packet [18]. An example of such a mode, likely to be adopted by NIST in the near future, is a counter mode [17]. Our implementation will be extended to permit such new modes as soon as they become adapted as draft standards.

Our architecture can be extended by adding additional outer-round pipeline stages, or implementing multiple instantiations of the same encryption/decryption unit, and using them for parallel processing of data. The total throughput in these extended architectures is directly proportional to the amount of resources (CLB slices, dedicated RAMs) devoted to the cryptographic transformations.

3.3 Round Key Module

The round key module consists of the 3-in-1 key scheduling unit and 16 banks of round keys. The banks of round keys are implemented using 8 Block SelectRAMs configured as two memories 256 x 64 bits. These memories permit storing up to 16 different sets of round keys, with 16 consecutive memory locations reserved for each set. Each set of subkeys may correspond to a different main key and a different security association.

The 3-in-1 key scheduling unit of Rijndael is shown in Fig. 5a. The operation of the circuit is described by formulas given in Fig. 5b. The unit is capable of computing two 32-bit words of the key material (w_i and w_{i+1}) per one clock cycle, independently



b)

For i mod Nk = 0

$w_i = w_{i-Nk} \oplus \mathrm{Sub}(\mathrm{Rot}(w_{i-1})) \oplus \mathrm{Rcon}_{i / Nk}$
$w_{i+1} = w_{i-Nk+1} \oplus w_{i-Nk} \oplus$
$\oplus \mathrm{Sub}(\mathrm{Rot}(w_{i-1})) \oplus \mathrm{Rcon}_{i / Nk}$

for Nk=8 and i mod Nk = 4

$w_i = w_{i-Nk} \oplus \mathrm{Sub}(w_{i-1})$
$w_{i+1} = w_{i-Nk+1} \oplus w_{i-Nk} \oplus \mathrm{Sub}(w_{i-1})$

otherwise

$w_i = w_{i-Nk} \oplus w_{i-1}$
$w_{i+1} = w_{i-Nk+1} \oplus w_{i-Nk} \oplus w_{i-1}$

Fig. 5. The 3-in-1 key scheduling unit of Rijndael: a) the main circuit, b) formulas describing the operation of the circuit

of the size of the main key. Since each round key is 128 bit long (the size of the input block), two clock cycles are required to calculate each round key. Therefore, our key scheduling unit is not designed for computing subkeys on the fly. Instead, all round keys corresponding to the new main key are computed in advance and stored in one of the memory banks. This computation can be performed in parallel with encrypting data using previous main key, therefore key scheduling does not impose any performance penalty.

4 Implementation of Triple DES

4.1 Basic Architecture

In order to realize the Triple DES encryption and decryption it is sufficient to implement only one round of DES, as shown in Fig. 6a. The multiplexers *mux1* and *mux2* permit loading new data block or feed back the result of the previous iteration. Only half of the data block is transformed in each iteration, and this transformation depends on a round key coming from the key module. The DES-specific transformation function *F* has been implemented as a combinational logic and directly follows the algorithm specification. The multiplexers *mux3* and *mux4* choose the right feedback for consecutive iterations. In the single DES implementation, these multiplexers would not be required, because the feedback is always the same. However, this is not the case for Triple DES because of the data swapping at the end of the last round of DES. This feature becomes important when switching between the first and the second, and between the second and the third DES encryption in Triple DES. Performing the Triple DES encryption or decryption of one data block in the CBC mode requires 48 clock cycles, exactly as many as the number of the cipher rounds.

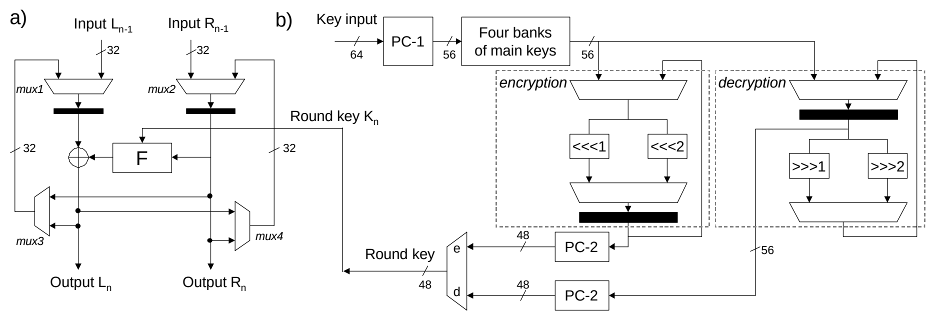


Fig. 6. Basic iterative architecture of Triple DES a) encryption/decryption unit, b) key scheduling unit

4.2 Round Key Module

The DES key schedule, which serves as a basis for the Triple DES key schedule, consists of very simple operations. Consecutive round keys are computed by rotating two halves of the main 56-bit key by one or two positions depending on the number of the round. The result of each next rotation goes through the Permuted Choice-2 function (PC-2), which selects 48 bits of a round key. Since DES key scheduling requires much simpler operations than encryption/decryption unit, it can be easily performed on the fly. This way only three 56-bit keys need to be stored on-chip. Our Triple DES key scheduling unit is shown in Fig. 6b.

Four banks of the key memories are placed at the input to the key scheduling circuit. Each bank contains three DES keys used by Triple DES. The user supplies 64-bit keys to the circuit, but only 56-bits of each key are selected by the Permuted Choice-1 function (PC-1) and stored in one of the memory banks. Each memory bank can hold all three keys required for performing Triple DES. All memory banks are built using dual-port memory, and can operate independently. They are organized in a way that permits writing new key to one of the banks, while any other bank may be used for the round key computations. The output of the round key memory goes to two simple circuits, one computes keys for encryption, the other for decryption.

4.3 Extended Architecture

We are currently in the process of developing an extended pipelined architecture of Triple DES. Our goal is to obtain throughput over 1 Gbit/s. Our approach is to fully unroll single DES and introduce pipeline registers between cipher rounds, as shown in Fig. 7. This leads to a capability of processing up to 16 independent data streams, which gives a throughput of around 1.5 Gbit/s. We should be able to maintain clock frequency at the similar or even greater level, since this architecture permits significant simplifications compared to the basic iterative architecture. Namely, multiplexers *mux3* and *mux4* are no longer required in any of the stages (see Fig. 6b), and key scheduling can be greatly simplified as shown in Fig. 7b.

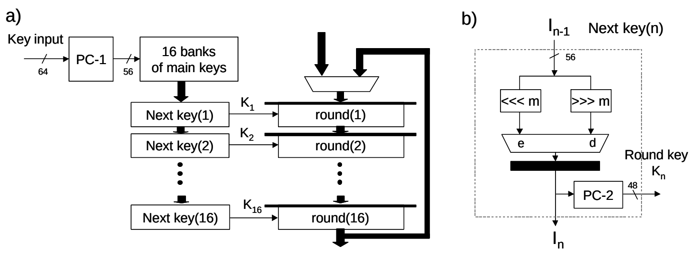


Fig. 7. Extended architecture of Triple DES: a) main circuit, b) next key module; the number of rotations m depends on the round number n, and can be equal to 0, 1, or 2

5. Testing Procedure

Our testing procedure is composed of three groups of tests. The first group is aimed at verifying the circuit functionality at a single clock frequency. The goal of the second group is to determine the maximum clock frequency at which the circuit operates correctly. Finally, the purpose of the third group is to determine the limit on the maximum encryption and decryption throughput, taking into account the limitations of the PCI interface.

Our first group of tests is based on the NIST Special Publication 800-20, which defines testing procedures for Triple DES implementations in ECB, CBC, CFB and OFB modes of operation [20]. This publication recommends two classes of tests for verification of the circuit functionality: Known Answer Tests (KATs), and the Monte-Carlo tests. Since, the Known Answer Tests are algorithm specific, we implemented them only for Triple DES. The Monte Carlo test is algorithm independent, so we implemented it for both Triple DES and Rijndael. The operation of this test is shown in Fig. 8. The test consists of 4,000,000 encryptions with keys changed every 10,000 encryptions. The ciphertext block obtained after each sequence of 10,000 encryptions is compared with the corresponding block obtained using software implementation. Software implementations of Triple DES and Rijndael from publicly available Crypto++ 4.1 library were used in our experiments.

The second group of tests was developed based on the principle similar to the Monte-Carlo tests. One megabyte of data is sent to the board for encryption (or decryption), the result is transferred back to the host, and downloaded again to the board as a subsequent part of input. The procedure is repeated 1024 times, which corresponds to encrypting/decrypting a 1 GB stream of data using CBC mode. Only

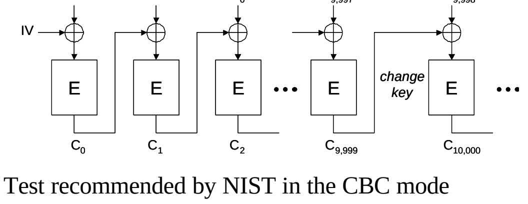


Fig. 8. Monte Carlo Test recommended by NIST in the CBC mode

the last megabyte of output is used for verification, as it depends on all previous input and output blocks. The transfer of data is performed by the DMA unit, so it takes place simultaneously with encryption/decryption. If the test passes, it is repeated at the increased clock frequency. The highest clock frequency at which no single processing error has been detected is considered the maximum clock frequency. In our experiments, this test was repeated 10 times with consistent results in all iterations.

The third group of tests is an extension of the second group. After determining the maximum clock frequency, we measure the amount of time necessary to process 4 GB of data, taking into account the limitations imposed by the 32 bit/33 MHz PCI interface. Since data is transmitted through the PCI interface in both directions (input and output), the maximum encryption/decryption throughput that can be possibly measured using this test is equal to 528 Mbit/s. This is a half of the maximum throughput in the regular operation of the FPGA accelerator, where only input data are transferred from the host to the accelerator card through the PCI interface, and the output is transferred from the FPGA card to the Ethernet daughter card.

6. Results

The results of static timing analysis and experimental testing for Rijndael and Triple DES are shown in Fig. 9.

For Triple DES in the basic iterative architecture, the maximum clock frequency is equal to 72 MHz according to the static analyzer, and 91 MHz according to the experimental testing using the SLAAC-1V board.

For Rijndael in the basic iterative architecture, the results for encryption and decryption are different, with decryption slower than encryption by about 13% in experimental testing. According to the timing analyzer, the maximum clock frequency for the entire circuit is equal to 47 MHz, with the critical path determined by the decryption circuit. In experimental testing, decryption works correctly up to 52 MHz, and encryption up to 60 MHz. However, we do not intend to change the clock frequency on the fly, therefore 52 MHz sets the limit for the entire circuit. The differences between the static timing analysis and experimental testing are caused by conservative assumptions used by the Xilinx static timing analyzer, including the worst case parameters for voltage and temperature prorating.

In Fig. 9b, the maximum throughputs corresponding to the analyzed and experimentally tested clock frequencies are estimated based on the equation:

$$\mathit{Maximum\_Throughput} = (\mathit{Block\_size} / \mathit{\#Rounds}) \cdot \mathit{Maximum\_Clock\_Frequency}. \quad (15)$$

Using formula (15), the maximum throughput of Rijndael in the basic iterative architecture for a 128-bit key is 521 Mbit/s based on the static timing analysis, and 577 Mbit/s based on the experimentally measured clock frequency. This result is expected to be further improved by optimizations of placement and routing. Taking into account our result, parallel processing of only two streams of data should be sufficient to obtain the speed over 1 Gbit/s. As a result, one stage of additional registers, R2a-c, was added to the basic iterative architecture in the extended

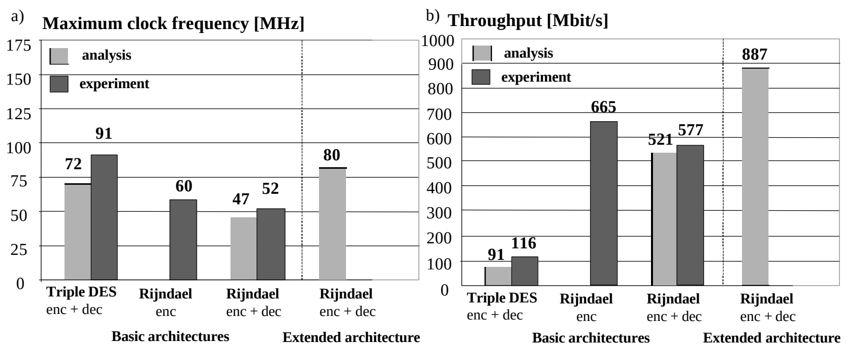


Fig. 9. Results of the static timing analysis and experimental testing for Rijndael and Triple DES a) maximum clock frequency, b) corresponding throughput

architecture as shown in Fig. 3. At this moment, we have been able to obtain a throughput of 887 Mbit/s for this extended pipelined architecture. Nevertheless, further logic and routing optimizations are expected to improve this throughput over 1 Gbit/s without the need of introducing any additional pipeline stages.

The worst-case throughput of Triple DES in the basic iterative architecture is 91 Mbit/s based on the static timing analysis, and 116 Mbit/s based on the experimentally measured maximum clock frequency, which translates to the 27% speed-up in experiment. Sixteen independent streams of data processed simultaneously should easily exceed 1 Gbit/s, leading to the extended architecture shown in Fig. 7.

The actual encryption and decryption throughputs, taking into account the limitations imposed by the PCI interface were measured using the third group of tests described in Section 5. The actual throughputs for DES, were equal to 102 Mbit/s for encryption, and 108 Mbit/s for decryption. The experimentally measured throughput for Rijndael was equal to 404 Mbit/s, and was the same independently of the key size, which means that this throughput was limited by the PCI interface. It should be noted that during the regular operation of the card, when no output is transferred back to the host memory, this throughput can be easily doubled and reach at least 808 Mbit/s.

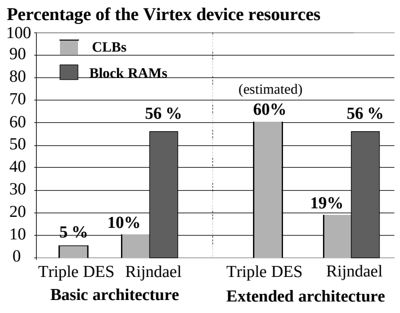


Fig. 10. Percentage of the Virtex device resources devoted to the encryption circuits

The total percentage of the FPGA resources used for the *basic* iterative architectures of Rijndael and Triple DES is 15% of CLB slices and 56% of BlockRAMs. The *extended* architectures of both ciphers, capable of operating over 1 Gbit/s, will take approximately 80% of CLB slices, and 56% of Block SelectRAMs. Only one Virtex XCV-1000 FPGA is necessary to assure the throughput of both ciphers in excess of 1 Gbit/s. Using two additional Virtex devices, and more complex architectures, the encryption throughput in excess of 3 Gbit/s can be accomplished. Our 64-bit/66 MHz PCI module will support this bandwidth.

7. Related Work

Several research groups developed VHDL implementations of Rijndael in Xilinx FPGAs [3, 6, 7, 12, 14], and Altera FPDs [8, 9, 10, 19]. A survey and relative comparison of results from various groups is given in [13]. All major results described in the aforementioned papers are based on the static timing analysis and simulation, and have not yet been confirmed experimentally.

The first attempt to validate the simulation speed of Rijndael through experimental testing is described in [9]. The test was performed using especially developed PCI card. Nevertheless, since the operation of the system appeared to be limited by the PCI controller, no numerical results of the experimental tests were reported in the paper.

As a result, our paper is the first one that describes the successful experimental testing of Rijndael and directly compares the experimental results with simulation.

8. Summary and Possible Extensions

The IPSec-compliant encryption/decryption units of the new Advanced Encryption Standard - Rijndael and the older encryption standard Triple DES have been developed and tested experimentally. Both units support the Cipher Block Chaining mode. Our experiment demonstrated up to 27% differences between the results obtained from testing and results of the static timing analysis based on Xilinx tools. These differences confirmed that the results based on the static analyzer should be treated only as the worst-case estimates.

The experimental procedure demonstrated that the total encryption and decryption throughput of Rijndael and Triple DES in excess of 1 Gbit/s can be achieved using a single FPGA device Virtex 1000. Only up to 80% of resources of this single FPGA device are required by all cryptographic modules. The throughput in excess of 3 Gbit/s can be accomplished by using two remaining FPGA devices present on the SLAAC-1V accelerator board. The alternative extensions include the implementation and experimental testing of other security transformations of IPSec, such as HMAC and the Internet Key Exchange protocol.

References

1. Advanced Encryption Standard Development Effort. http://www.nist.gov/aes
2. Blaze M., Diffie W., Rivest R., Schneier B., Shimomura T., Thompson E., and Wiener M.: Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security http://www.counterpane.com/keylength.html
3. Chodowiec P., Khuon P., Gaj K.: Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining. Proc. ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays, FPGA'01, Monterey, Feb. 2001, 94-102
4. Davida G. I. and Dancs F. B.: A crypto-engine, Proc. CRYPTO 87, (1987) 257-268
5. Daemen J. and Rijmen V.: Rijndael: Algorithm Specification. http://csrc.nist.gov/encryption/aes/rijndael/
6. Dandalis A., Prasanna V. K., Rolim J. D.: A Comparative Study of Performance of AES Final Candidates Using FPGAs. Proc. Cryptographic Hardware and Embedded Systems Workshop, CHES 2000, Worcester, MA, Aug 17-18, 2000
7. Elbirt A. J., Yip W., Chetwynd B., Paar C.: An FPGA implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. Proc. 3rd Advanced Encryption Standard (AES) Candidate Conference, New York, April 13-14, 2000
8. Fischer V.: Realization of the Round 2 AES Candidates Using Altera FPGA. Submitted for 3rd Advanced Encryption Standard (AES) Candidate Conference, New York, April 13-14, 2000; http://csrc.nist.gov/encryption/aes/round2/conf3/aes3papers.html
9. Fisher V.: Realization of the RIJNDAEL Cipher in Field Programmable Devices. Proc. of DCIS 2000, Montpellier, Nov. 2000, 312-317
10. Fisher V., Drutarovský M.: Two methods of Rijndael implementation in reconfigurable hardware. Proc. of CHES 2001, Paris, 2001
11. Frankel S., Kelly S., Glenn R.: The AES Cipher Algorithm and Its Use with IPSec. Network Working Group Internet Draft, November 2000, (work in progress) available at http://ietf.org/html.charters/ipsec-charter.html
12. Gaj K., Chodowiec P.: Comparison of the Hardware Performance of the AES Candidates Using Reconfigurable Hardware. Proc. 3rd Advanced Encryption Standard (AES) Candidate Conference, New York, April 13-14, 2000
13. Gaj K. and Chodowiec P.: Hardware performance of the AES finalists survey and analysis of results, Technical Report available at http://ece.gmu.edu/crypto/publications.htm
14. Gaj K. and Chodowiec P.: Fast Implementation and Fair Comparison of the Final Candidates for Advanced Encryption Standard Using Field Programmable Gate Arrays, Proc. RSA Security Conference - Cryptographer's Track, April 2001
15. IP Security Protocol (ipsec) Charter - Latest RFCs and Internet Drafts for IPSec, http://ietf.org/html.charters/ipsec-charter.html
16. Jones M., Athanas P. *et al.*: Implementing an API for Distributed Adaptive Computing Systems. IEEE Workshop on Field-Programmable Custom Computing Machines, Napa Valley, CA, Apr. 1999, 222-230
17. Lipmaa H., Rogaway P., Wagner D.: CTR-Mode Encryption, Public Workshop on Symmetric Key Block Cipher Modes of Operation. Oct. 2000, Baltimore, MD, http://csrc.nist.gov/encryption/modes/workshop1/
18. Modes of Operation. http://csrc.nist.gov/encryption/modes/
19. Mroczkowski P.: Implementation of the Block Cipher Rijndael Using Altera FPGA. Public Comments on AES Candidate Algorithms - Round 2. http://csrc.nist.gov/encryption/aes/round2/pubcmnts.htm.
20. NIST Special Publication 800-20, Modes of Operation Validation System for the Triple Data Encryption Algorithm, National Institute of Standards and Technology (2000)
21. Secure Hash Standard Home Page. http://csrc.nist.gov/cryptval/shs.html
22. Smith R. E.: Internet Cryptography, Addison-Wesley (1997)

FPGA-Based Sonar Processing

Paul Graham and Brent Nelson
Department of Electrical and Computer Engineering
Brigham Young University, Provo, UT

1 Introduction

Field programmable gate arrays (FPGAs) have been used for many computational tasks since their invention [1, 2, 3, 4, 5]. In much of the work to date, FPGAs have been found to be reasonable alternatives to custom hardware (ASICs) or software implementations of applications—they provide speed-ups over software through hardware specialization while still providing the flexibility to adapt the hardware to changing application needs [6].

A large percentage of the problems approached in the research community have focused on smaller computational problems–problems for which one or a handful of general-purpose processors (GPP) or digital signal processors (DSPs) could be used to compute the results in a reasonable amount of time, though more slowly than hardware solutions to the same problems. In this paper, we discuss an application—sonar beamforming—which can require tens or hundreds of GPPs or DSPs to provide the results in real-time.

In studying sonar beamforming, we have found that some FPGA-based computing solutions to this signal processing technique can outperform DSPs, even when beamforming performs computations which appear to be naturally suitable to DSPs. Additionally, we have found that, coupled with a high-bandwith interconnection network, FPGA-based systems can function well as multiprocessor systems, especially, when taking advantage of the FPGAs' abilities to perform custom, application specific I/O functions.

Below, we will first introduce the sonar beamforming application and its computational requirements. Next, we will provide one of our designs for the beamforming application using FPGA-based custom computing machines (CCMs), including a discussion of individual processor architectures and overall system architectures. Following this introduction to sonar beamforming on FPGAs, we will provide a comparison of sonar processing on FPGA-based and DSP-based multicomputers. The final sections of the paper will then outline future work that we plan to complete in this area as well as a summary of our conclusions.

2 Conventional Beamforming

For the purposes of this paper, we will be concentrating on conventional sonar beamforming, as opposed to adaptive beamforming. Additionally, we are restricting our discussion to time-domain beamforming because of the maturity of our work in that area as well as to reduce the scope of this paper.

In general, beamforming is a spatial filtering operation performed on the data received by an array of sensors, such as antennas, microphones, or hydrophones. It provides a system with the ability to "listen" directionally even when the individual sensors in the array are omnidirectional. Beamforming not only causes a system to be more sensitive to signals coming from a specific direction, but it also attenuates the noise and interferences coming from other directions.

One method used to perform beamforming in the time domain is delay-sum, or time-delay, beamforming. In this method, the spatial filtering results from the coherent (in-phase) summing of the signals perceived by the different sensors in the array. A signal's propagation delay from sensor to sensor can be calculated through knowledge of the speed of the signal through the medium, the distance between the sensors, and the signal's direction of propagation. With this information, signals received by the array coming from the direction of interest can be summed together in-phase by taking the appropriately delayed samples from a sample history buffer for each sensor. Signals approaching from directions other than the direction of interest are not coherently summed, thus, they will not have a maximum response from beamforming, though they may have some response. Delay-sum beamforming also has the characteristic that the beams formed are “broadband” since they can be sensitive to a relatively large range of frequencies as opposed to being tuned to specific frequencies. Despite or even because of its simplicity, delay-sum beamforming is still commonly used in many sonar applications.

The following pseudo-code represents the delay-sum beamforming calculation for a single beam:

```
formBeam(b) {
  response = 0;
  for (s=0;s<numSensors;s++)
  response = response +
            shade[b][s]*dataSamples[s][delayFunction(b,s)];
}
```

The calculation is basically a multiply-accumulate (MAC) operation which applies a windowing function, represented by the *shade* array, to the appropriately delayed versions of the received signal for all of the sensors. The *dataSamples* array represents a fixed-size buffer which holds a running history of the last N samples received by each sensor. The function *delayFunction* returns the location of the sample to sum, using the beam’s direction and the sensor’s position to ensure that signals coming in the beam’s direction are coherently summed. Notice that the shade, or windowing, values are also dependent on the beam direction and sensor position; this provides the designer with the ability to determine the filter’s spatial response, called the *beam response*, for each beam direction, including the ability to cancel out known interferences.

Though, the calculation itself is trivial, the demands of performing this calculation for thousands of beams with real-time processing constraints greatly increases the amount of hardware required for the application. For example, if an array of 400 sensors is being sampled at a frequency of 2 kHz and if 10,000 beams must be formed, then the computation requires a processing rate of:

$$\begin{aligned} R_{calculations} &= 2000\, kHz \cdot 400\, sensors \cdot 10000\, beams \cdot 2\, ops && (1) \\ &= 16x10^9\, operations\, per\, second && (2) \end{aligned}$$

So, approximately 16 $billion$ operations—8 $billion$ multiplies and 8 $billion$ additions—must be performed per second. With a calculation rate of one arithmetic operation per cycle, it would require 80 200 MHz processors to calculate at this rate, ignoring all processing overhead [7]. This example problem is similar but a bit smaller than a few applications that we have worked on.

For a detailed treatment of beamforming and its many forms, we refer the reader to [8], [9], and [10].

3 FPGA Implementations of Time-Domain Beamforming

3.1 Processor Architectures

The pseudocode in Figure 1 describes the particular implementation of delay-sum beamforming we implemented on FPGA-based hardware. First, we must decide on the location of the sample which must be summed for each sensor in a given beam calculation. This is generally done through a look-up table, since the location of the sample in memory for a given beam and sensor is not some simple function. This complexity is due to the fact that the sensor arrays may not have a simple shape, the wave fronts themselves may not be planar (i.e., near-field sources), and/or the fact that the delays between succesive sensors are not generally some integral number of sample periods. Once the sample’s location is known, it is then retrieved from memory and multiplied by the shade factor. The result of this produc is accumulated for all sensors in the beam.

For real-time operation, the beamforming hardware must calculate all of the beams of interest for each set of new samples; in other words, as implied by the computational requirements discussed earlier, all beams must be calculated at the sample rate for the sensors. Again, this is why delay-sum beamforming can be so computationally expensive, especially when hundreds or thousands of beams must be formed.

```
beamForm(beam) {
  acc = 0;
  for (s=1 to numSensors) {
    delay = delayLookup[beam][s];
    acc = acc + samples[delay]*shade[beam][s];
  }
}
```

Figure 1: FPGA-based Beamforming Pseudocode

The sample data for the designs discussed in this paper are assumed to be 12-bit fixed point values. The FPGAs-based beamformer designs use the data at full 12-bit precision and account for bit growth during the accumulations and multiplies, so the results are also provided in full precision. Additionally, for the example problem having $10,000$ beams, 400 sensors, and a $2\,kHz$ sample rate, the number of samples stored per sensor is assumed to be about 650, which is representative of the number of samples required for linear sonar arrays having characteristics similar to that of our example.

We have created several different delay-sum FPGA-based beamformer designs, but all generally have the structure shown in Figure 2. The design discussed in the rest of this paper is similar to the general structure given except both the *delayLookup*, or sample locations, and shade values are stored together in a single memory instead of being in two separate memories. More specifically, the sample address and shade factors for each sensor and beam are packed into a single memory word to reduce memory requirements and to simplify memory addressing.

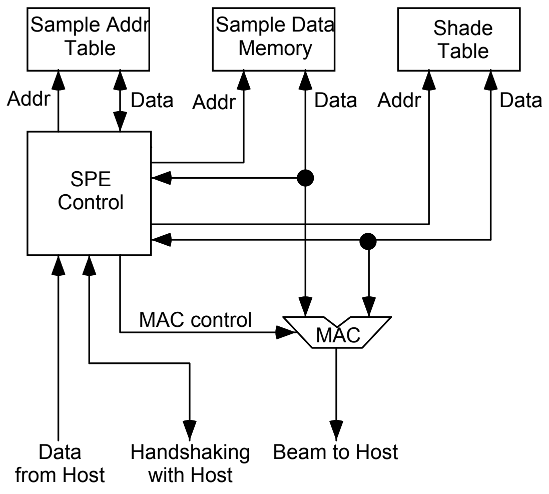


Figure 2: Beamforming Processor Block Diagram

The FPGA-based beamformer for our example problem is quite small, under 400 Xilinx XC4000XL CLBs and can execute at frequencies on the range of 40 to 50 MHz with pipelining on “-1” speed grade Xilinx XC4000XL FPGAs. As a result of the pipelining, the processors are able to complete one beamforming multiply-accumulate (MAC) per cycle. The memories used for the designs are 256Kx16 SRAMS, four of which surround each FPGA in the system. This permits 2 beamforming processors to be placed on each FPGA. For additional information on the structure of the FPGA boards to which the beamforming algorithm were mapped, see the discussion below on the system-level architecture.

As a final note on FPGA-based delay-sum beamformers, all of the beamformers that we have designed have a structure similar to that of Figure 2, as we mentioned earlier. The main differences among our beamformer designs result from the fact that we were able to take advantage of array symmetries for some arrays to reduce the storage requirements of the beamforming processors. Besides the storage requirements, this also impacts the complexity of the address generation logic for the sample location/shade value look-up. For instance, in one particular design for a spherical sonar array, the sample addresses and shade values are not even stored in local memory but are broadcast to all of the FPGA-based beamformers since all processors use the same values for different beams due to array symmetry. For this spherical array design, the memory requirements for each processor reduced to only one external SRAM, allowing us to place four processors per FPGA.

3.2 System Architecture

The overall target platform for our beamforming designs is based on the FPGA computing board described in Figure 3—a custom computing machine design proposed for use with several DARPA-related projects. Each board has an interface to a fast, low-latency network known as Myrinet [11]. The connections in the network are made from point to point between two nodes, as opposed to a bus structure; each connection is capable of providing two 1.28 Gbit/s ($\approx$ 160 MB/s) communications channels—one for incoming data and one for outgoing data. Myrinet, then, provides the interconnection network for large, multi-board systems.

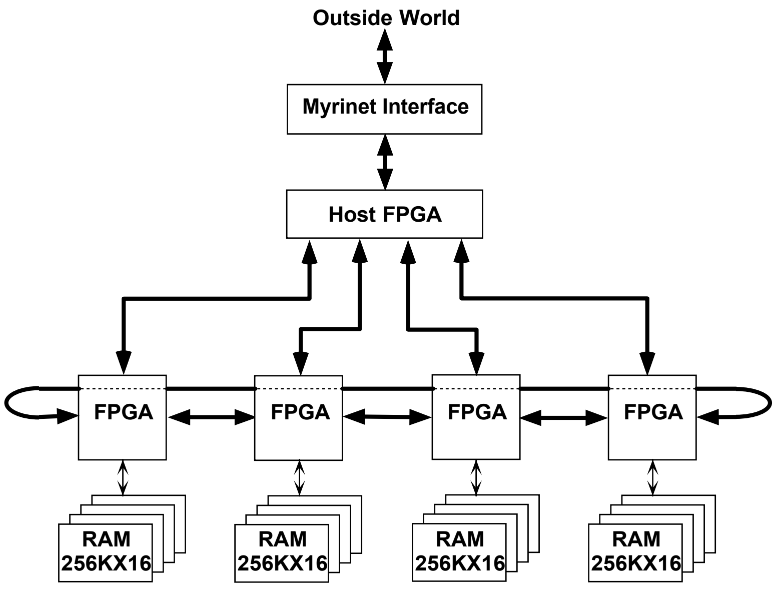


Figure 3: FPGA Multicomputer Board

The board includes one Myrinet interface processor, one "host" FPGA, and four FPGA processing elements (PEs). The PEs are connected in a simple ring, each PE connecting to two neighbors. For the transmission of data and the configuration of the four PE FPGAs, the host FPGA has a separate bus to each PE. Additional buses (not shown) allow the PEs to connect directly to neighboring computing boards or to interface with other digital equipment.

Several options exist for organizing the beamforming calculation across processors. The first option is to have each beamformer calculate the beam response for some subset of the total beams. As each beam is calculated by the PEs, the results are sent to the host FPGA, which forwards the data over Myrinet to the network's main beamforming host for further processing and display. We call this the *beam-oriented* approach to distributing the beamforming application across the processors. This approach generally requires the most processor storage since all of the sensor data must reside at each processor, but it also requires the smallest amount of interprocessor communication of the beamforming organizations.

Another organization option is to have each beamforming processor perform the MACs for a subset of the sensors but for all of the beams, thus, creating only partial beam responses for all beams. The partial beam responses are then summed externally to the processors themselves to finish the beam response calculations. We describe this organization as the *sensor-orfiented* approach. For example, if there are 400 sensors, $10,000$ beams, and 100 beamforming processors, each beamforming processor might be responsible for performing the MACs for only 4 sensors of every beam. To complete the beamforming calculation, a network of accumulators are used to accumlate the partially-formed beams; these accumulators may reside on the FPGAs with the beamformers and/or on a separate FPGA computing board. This architecture has the smallest memory requirements and the largest amount of interprocessor communication. The smaller memory requirement is the result of storing only sample data for a few sensors on each processor, as opposed to the entire data set.

The last main organization is simply a combination of the two approaches, in other words, each beamforming processor may perform the MACs for a subset of the beams and for a subset of the sensors. This provides a way of balancing the memory requirements of each processor with the amount of interprocessor communication required.

3.2.1 Memory Size Comparison

To contrast the memory requirements between the beam-oriented and sensor-oriented approaches, we will apply these two organizations to our example beamforming problem with 400 sensors and $10,000$ beams. Before comparing the memory requirements of the two approaches, we must decide how many FPGA-based processors are required to perform the example beamforming problem in real-time. Since the example problem requires a MAC rate of $8x10^{9}$ MACs/second for real-time calculation and a 40-MHz FPGA-based beamformer can perform $40x10^{6}$ MACs/second, we know that problem would require about 200 FPGA-based processors, or 25 FPGA computing boards with two beamformers per FPGA.

For the beam-oriented organization, we want to distribute the processing of the $10,000$ beams evenly, so each of the 200 processors must calculate 50 complete beams. Assuming a sample history of 650 samples per sensor, each processor in this organization must store $650\, samples \cdot 400\, sensors = 260,000\, samples = 254x2^10 samples$. Since each sensor sample is a 12-bit value, the entire history can be stored in one of the external 256Kx16 SRAMs attached to the FPGA. The total number of sample location/shade value pairs that each processor must store is: $400\, sensors \cdot 50\, beams = 20,000\, value\, pairs$. Assuming that each sample location/shade value pair can be stored in a single 16-bit word, the total amount of memory required per processor is then $280,000$ 16-bit values.

For a sensor-oriented organization using 200 processors, each processor would only calculate the partial products for two sensors. Thus the amount of sensor sample data stored locally in each processor is $650\, samples \cdot 2 sensors = 1300 samples$. As for the number of address/shade values required, each processor must store: $2\, sensors \cdot 10,000\, beams = 20,000 pairs$. This amounts to a total memory requirement of $21,300$ 16-bit values.

3.2.2 Communication Bandwidth Comparison

Next, we will contrast the communication bandwidth required by the two extremes in organization for the example beamforming problem. The main concern and constraint is that the communication across any point-to-point Myrinet link must not exceed 160 MB/s in either direction.

For the beam-oriented approach and the example beamforming problem, each beamformer calculates 50 beams per sample period. Since two beamformers fit per FPGA, a FPGA computing board has eight beamformers and, consequently, each board calculates 400 beams per sample period. Assuming that each beam response is represented as a 32-bit word, the communication bandwidth requirement from each board must be $400/, beams \cdot 4\, bytes/beam \cdot 2\, kHz = 3.2x10^6\, bytes/sec$, which is well under the 160 MB/s unidirectional bandwidth limit of a single Myrinet point-to-point link. The link which will have the most traffic is the one which leads to the network's main host. This host must receive the data from all beamformers every sample period. Thus the required input bandwidth to this node is $10,000\, beams \cdot 4\, bytes/beam \cdot 2\, kHz = 80x10^6\, bytes/sec$, which is about half of the unidirectional maximum throughput of a Myrinet link. So, the Myrinet network provides easily enough bandwidth for the beam-oriented beamforming organization.

For the sensor-oriented approach, each processor produces a partial beam sum for two sensors only but for all of the beams. Assuming that the remaining logic on the PE FPGAs and the host FPGA can be used to sum the partial beams on board for each beam, then each FPGA processing board produces a single partial beam for each of the 10,000 beams. Also, assume that because of the smaller number of sensors per partial beam, the partial beam can be represented in three bytes. Consequently, each of the 25 beamforming FPGA boards requires an output communications bandwidth of $10,000\, partial\, beams \cdot 3\, bytes/beam \cdot 2\, kHz = 60x10^6\, bytes/second$, which is much greater than the $3.2x10^6\, bytes/second$ per board in the beam-oriented organization.

For completeness, let us finish the communications bandwidth analysis for the sensor-oriented organization. Note that the total bandwidth from the beamforming FPGA boards to the accumulating FPGA boards for the sensor-oriented organization is $25\, boards \cdot 60x10^6\, bytes/sec\, per\, board = 1.5x10^9 bytes/sec$. Clearly, to perform the accumulations, the network loads point-to-point must not exceed the 160 MB/sec unidirectional limit. If we assume that only 80% of peak unidirectional throughput is sustainable and the network load is balanced across several accumulating FPGA boards, then the number of FPGA accumulating boards required based on network throughput is:

$$boards_{accumulating} = \frac{1.5x10^9\, bytes/sec}{.80 \cdot 160\, MB/s\, per\, board} \tag{3}$$

$$\approx 12 boards \tag{4}$$

So the total number of FPGA computing boards would be around 37, including both beamforming and accumulating boards. If each Myrinet link is at 80% of capacity, then each accumulating board is receiving 128 MB/sec of partial beam results. With three bytes per partial beam, this translates into about $44.7x10^6 partial\, beams/sec$. The accumulating board should easily handle this accumulation rate since just two FPGAs operating at 40 MHz should be able to accumulate a sample per cycle, which translates into an accumulation rate of $80x10^6\, accumulations/sec$ for just two FPGAs. Lastly, note that the communication bandwidth to the network's host is still $80x10^6\, bytes/sec$ since the number of beam responses communicated to the main host is the same as in the beam-oriented organization. So, the bandwidth of the link to the host should still be adequate.

Considering the number of FPGA computing boards and the amount of communication bandwidth required for each organization, the beam-oriented approach is the clear choice for our example beamforming problem, especially, since each FPGA has enough SRAM to support two beam-oriented processors. With this information, we will assume that the FPGA-based beamformers used in the following comparisons with DSPs are in the beam-oriented system organization.

4 Comparison with DSP Implementations

For the sonar beamforming application area, one of the main competing technologies would be digital signal processors which have support for multi-DSP systems. DSPs are a natural choice for delay-sum beamforming since they have been optimized to perform multiply-accumulate operations efficiently. Also, since beamforming will require many DSPs to work in parallel to perform the calculations in real-time, support for multi-DSP systems is important also. A handful of DSPs have been designed with multiprocessing in mind, a few of which are the Analog Devices SHARC DSP family and TI's TMS320C4x DSP family. The comparisons given in this section will be with the SHARC family since it appears to be more prevalent for large-scale multiprocessing DSP applications.

This section will begin with a brief introduction of the SHARC DSP and how well it performs delay-sum beamforming. With this background, FPGA-based systems will be compared with SHARC DSP systems based on the performance per processing node, the cost/performance of the solutions, and their ability to adequately support the interprocess communication required for delay-sum beamforming. The comparison concludes by contrasting the two application development methods for the platforms.

4.1 The SHARC DSP

As a brief introduction, the Analog Devices' Super Harvard Architecture Computer (SHARC) DSP has been designed to perform several digital signal processing tasks efficiently. SHARC DSPs have a three-stage pipeline and execute each supported arithmetic operation in a single cycle. Additionally, with its instruction caching scheme, the SHARC has a peak computation rate of up to three single-precision floating-point or three 32-bit fixed-point operations per cycle, assuming all of the operands are residing in its register set. The SHARC DSPs also include on-chip SRAM to provide fast access to operands.

As an example of the SHARC's computational performance, the DSP can execute a floating-point multiply and add in a single instruction, Thus, it can complete a floating-point multiply-accumulate (MAC) every cycle. To achieve this, the multiply and add operations are "pipelined" in software—in the first cycle, a value is multiplied with a scaling factor. In the next cycle, the scaled value is accumulated. So, every cycle the SHARC can carry out a new multiply while accumulating the result of the last scaling. For fixed point numbers, the MAC and one of a restricted few ALU operations can be performed in a single cycle.

To support the peak computational capacity of the SHARC processor, the DSP must have an efficient memory system. Each SHARC has two equally sized banks of dual-ported SRAM on chip, providing fast data accesses to two operands per cycle. Being a modified Harvard architecture processor, one of the on-chip memories stores program and data while the other is exclusively for data storage. Depending on the SHARC model, the total amount of on-chip SRAM ranges from 128KB to 512KB. One of the ports to each memory can be accessed by the DSP's core while the SHARC's external I/O interfaces have access to the second port, allowing for transparent access to the on-chip memories without disturbing the operation of the DSP core.

Two operands can be fetched from memory each cycle–one from the combined program/data memory and the other from the data-only memory. With this two-operand fetch, the bus to the program memory is being used to either load or store an operand, so the SHARC cannot access program memory simultaneously for the next instruction. The SHARC cleverly uses a two-way set-associative, 32-instruction cache to account for this situation. When this conflict for program memory first occurs, the SHARC will go to the cache to see if the next instruction resides in the cache. Being the first occurrance of the conflict, the instruction will not be in the cache and a cache miss occurs; the fetch for the instruction will occur after the data access from program/data memory and the instruction will be loaded into the cache. Assuming the instruction does not get replaced in the cache, the next time the instruction and two data transfers to or from memory coincide again, the instruction is simply retrieved from the cache, allowing unhindered access to the two internal SHARC memories. This means that the DSP can sustain a floating-point MAC per cycle as long as it has new operands to work on and the instruction is cached.

As an additional memory feature, the SHARC has two address generators, one for program/data memory and the other for data-only memory. The address generators provide support for circular buffer addressing, bit-reversed addressing (for FFTs), and several indirect and direct memory access modes, including a mode for addressing memory locations at regular intervals (strides). Specifically, the address generators are intended to support common DSP operations such as FIR filters and FFTs.

Lastly, the SHARC DSP family has been designed to support shared-memory multiprocessing as well as the dataflow, SIMD, and MIMD multiprocessing styles. Up to six SHARC processors plus a host processor can be combined in a single multiprocessing cluster for shared-memory multiprocessing. In this model, the host, the SHARCs, and external memory share a common cluster bus. Moreover, the internal memories of all of the SHARCs in the cluster are mapped into the global memory space of the array and are accessible by the host and the other SHARC DSPs. All accesses to the SHARC internal memories over the multiprocessor bus can be done transparently to the SHARCs' processing cores since their memories are dual ported, as mentioned before. To perform interprocessor communication, a SHARC in the cluster gains control of the bus (i.e., becomes the bus master) and can then read or write to the internal memories and registers of the other SHARCs directly or it can setup the slave SHARCS' DMA controllers to perform the data transfers. The SHARC multiprocessing architecture also supports a broadcast write where the bus master can write to the memories of all of the SHARCs in the array.

Dataflow, SIMD, and MIMD styles of processing are supported through the 6 bi-directional nibble-wide link ports which can be attached to individual, neighboring SHARC processors. Despite being only nibble-wide, each link port can actually transmit a byte per cycle, meaning that with typical SHARC clock rates of 40 MHz, the link port can transfer 40 MB/s. All six link ports can operate simultaneously.

The multiplicity of features provided by the SHARC makes it a common choice for large multiprocessing DSP applications. Despite all of these features, we will show that the SHARCs do not handle all DSP-like operations effectively, even when they should be a natural fit for DSPs.

4.1.1 Delay-Sum Beamforming on SHARC DSPs

The basic operations which the DSP must perform for the delay-sum beamforming operation are the following:

- A sample address table lookup for each sensor in a specific beam.
- A multiply-accumulate operation which multiplies the sensor sample values with the appropriate shading factors, accumulating the result for all of the involved sensors.

This is exactly what the beamforming pseudo-code provided in Figure 1 above represents.

Since the SHARC can perform a floating-point multiply and addition in a single cycle, the SHARC should be able to perform the multiply-accumulate operation quite efficiently, assuming the data can be provided at an adequate rate. Unfortunately, because of limitations in the address generation, a delay-sum beamforming MAC cannot be performed every cycle or every other cycle. At a minimum, a beamforming multiply-accumulate takes three cycles.

Below is an example of the inner-loop assembly code required for this operation. As a note of clarification, the f registers are single-precision floating point registers, the m registers are the offset registers found in the address generators, and the i registers are the base address registers of the address generators. Additionally, addressing to the data-only memory is done through the calls such as $f1 = dm(i1, m1)$, while program/data memory accesses are performed instructions such as $f9 = pm(i9, m9)$. The last detail worth mentioning is that memory accesses listed with the base (i) register first are using post-modify addressing where address formed is the sum of the base and offset registers and the base register is also loaded with the resulting sum. Pre-modify addressing is where the offset (m) register appears as the first argument; the address formed is again the sum of the the offset and base registers, but the base register in this case retains its original value after the memory access.

```
beamForm:
     m1=pm(i8,m8); /* sample loc table lookup from program/data memory */
     f4=pm(i9,m9); /* shade table lookup from program/data memory */
     f2=dm(m1,i1); /* sample history lookup from data-only memory */
     /* loop for beamforming MACs */
     lcntr = numSensors, do LoopEnd until lce;
          m1=pm(i8,m8); /* sample loc table lookup */
          /* parallel multiply, add, shade table lookup */
          f8=f2*f4, f12=f8+f12, f4=pm(i9,m9);
LoopEnd:  f2=dm(m1,i1); /* sample history lookup */
```

There are many reasons why the beamforming MAC requires three cycles to execute. First, the sample location table lookup takes a cycle, but cannot be performed in the same instruction as the floating-point multiply and add since it is modifying an address generator register (*m1*) as opposed to a general-purpose register such as *f2*. Additionally, the location table look-up cannot be performed concurrently with an access to the sample history data in the data memory since it is modifying a register in the data-only memory's address generator, i.e.,*m1*. The shade value lookup, the multiply, and add all can be done in a single cycle. The third cycle performs the look-up of the sensor sample data, but this cannot be done in conjunction with with the multiply, add, and shade look-up for two reasons: 1. the data-only memory address generator cannot form an address the cycle following any modification of its internal registers (e.g., *m1*). 2. the only type of memory operations allowed concurrently with multiply and ALU operations are those using post-modify addressing (*pm(i9,m9)*) as opposed to pre-modify addressing (*dm(m1,i1)* or *pm(m15,i15)*). So, a three cycle loop is required to perform each delay-sum beamforming MAC, a few more instructions than you might expect for this type of operation.

As far as memory efficiency is concerned, since the instruction cache is a two-way set-associative cache with 32 entries, those instructions fetches in the loop which conflict with accesses to the program/data memory can be easily cached. Because of the three-staged pipeline of the processor (fetch,decode,execute), both the fetches for the first and last instructions within the loop coincide with accesses to program/data memory and are held in the SHARCs instruction cache. The second instruction is not cached.

Now, you might ask, "Since DSPs are designed to do MACs efficiently, why does it take three cycles to perform a beamforming MAC?" Basically, the SHARCs' address generators and memory interfaces cannot support a one-

or two- cycle beamforming MAC. First, the signal delay between sensors for an arbitrary signal is generally not an integral number of sample periods, consequently, the DSP's ability to stride regularly through memory with its address generators cannot be effectively used for performing the fetching of sensor data for a beamforming MAC. In our beamforming example, the sample delays for each sensor and beam are generally real values which have been rounded appropriately to the nearest integer, so the algorithm employs a table look-up to find the proper sensor sample address. Additionally, when the wavefronts are assumed to be curved (near-field sources) or the receiving arrays are not linear (e.g,. spherical), the signal delays from sensor to sensor are known but nonlinear; in this situation, the appropriate sensor sample address is stored in a table for computational efficiency. (More sophisticated beamforming schemes also include an interpolation to more accurately estimate the values in between samples to compensate for non-integer signal delay values between sensors.)

4.2 Performance

In this section we want to compare the performance of SHARC DSPs with FPGA-based processors for the delay-sum beamformer. First, we will discuss raw performance issues and then the cost/performance of the two technologies.

4.2.1 Performance Comparison

For the beamforming example mentioned above (10,000 beams, 400 sensors, and $2kHz$ sample rate), the computation rate must be $8x10^9$ MACs/second. Since a SHARC DSP executing at 40 MHz (the highest current clock rating) would have a peak *beamforming* MAC rate of $\frac{40}{3}x10^6 = 13.3x10^6$ MACs/second, this particular beamforming operation would require about 600 SHARC processors for real-time operation, not including any host processors needed for SHARC shared-memory multiprocessing and ignoring interprocessor communication and other types of overhead.

Additionally, for the data to fit in the relatively small memory of the SHARC ADSP-21060, each processor must perform the MAC operations for 100 sensors and 200 beams. This is a combination of the sensor- and beam-oriented approaches mentioned earlier. Normally, for a fully beam-oriented approach, the computation would entail performing the complete MAC operation for 400 sensors and 50 beams; this would require 160 KB for the shade and delay values stored as single-precision floating-point values and an additional 1022 KB for the sensor data also stored in a single-precision format. Since the ADSP-21060 has only 512 KB of on-chip SRAM, this is not an acceptable memory configuration because it would require off-chip memory accesses, leading to contention for memory on the SHARC cluster's shared bus. By having each each SHARC DSP perform 200 beam calculations using only 100 sensors apiece, the amount of memory for sensor sample data is reduced to about 255 KB of data, while the memory required for shade and delay values is still 160 KB. As a result of this memory organization, no memory external to the SHARCs is required. (Note that 32-bit fixed-point formats are also possibilities, but that does not alleviate any of the storage problems. Also, the SHARC's special 16-bit floating point does not provide enough precision for the calculations, in general.)

The cost of this reorganization of the algorithm is that some processor (may be the host processor in each cluster) must accumulate the values from 4 SHARC processors to obtain 200 fully-formed beams. This is relatively little overhead ($\approx$800 cycles) considering that the multiply-accumulate time for each set of sample data is 20000 cycles and the accumulation by the host can be done in parallel with the four beamforming SHARCs without disturbing any calculations. Additionally, the placement of new sample data into the DSPs memories can be done transparently by the host.

In contrast to the SHARC implementation, the FPGA-based beamformers can perform a beamforming MAC every cycle through pipelined operation and special address-forming logic. Thus, 200 FPGA-based beamformers executing at 40 MHz are all that are required to perform the delay-sum beamforming calculation. At less than 400 CLBs per processor using 12-bit data samples, two beamformers can fit in a Xilinx 4028XL having four external 256Kx16 SRAMS. In this case, each FPGA can perform the computation of six SHARC DSPs. Having more processors per chip is mainly limited by the number of user I/O pins are available. For example, using a Xilinx XC4085 having 448 user-definable I/0 pins, 10 256Kx16 SRAMs could be attached to the FPGA while still leaving 88 other pins for inter-FPGA communication purposes, meaning that 5 FPGA-based beamformers could be embedded in the 4085 with only 30% overall logic utilization; a single 4085 operating at 40 MHz (assuming that is possible) would have the performance of 15 SHARC DSPs for our beamforming application.

To be fair, the FPGA-based beamfomers are performing lower precision arithmetic (12-bit, 18-bit, and 27-bit) than the SHARC, but neither single-precision floating point nor 32-bit fixed-point is not required for the arithmetic. The FPGA designs perform the calculations according to the precision of the data and allow for bit growth in the calculations so that results maintain full-precision. Consequently, the FPGA-based design fits the application requirements precisely without using larger-than-necessary data formats for calculations as the SHARC does. Additionally, as mentioned before, the FPGA design makes the simplification that the sample address and shade values are packed into a single 16-bit word (10-bits for the address value and 6-bits for the shading factor). How much range is actually required for the shading factor varies and 6-bits may be enough, in some cases. Actually, the 6-bit shading factor can be used as a look-up into an on-chip memory which stores the shade values with greater precision since plenty of on-chip logic is available to create these small memories.

As another example of the computational advantages of FPGA-based beamformers over a multi-DSP solution, take the delay-sum beamforming required for the spherical array mentioned briefly above. A single FPGA can act as four independent 40-MHz single-cycle beamforming processors, while a 40-MHz SHARC DSP can serve as a single three-cycle beamforming processor. In this case, a single FPGA can provide the computational power of 12 DSPs. The FPGA required for the FPGA processors is quite a bit larger, a Xilinx 4052XL; this is required so that enough routing and logic resources are available to reach a 40 MHz processing rate.

These six- and twelve-fold computational advantages enjoyed by the FPGA implementations of beamforming clearly stem from the increased parallelism enjoyed by the FPGAs. A few simple characteristics of the algorithm and the FPGA implementation contributed to this increased parallelism. First, the beamforming MAC operation is fairly simple and easily pipelined. As a consequence, the FPGA design can operate at speeds comparable to that of the SHARC DSPs. Also as a result of the simplicity of the operation, the individual beamformers are quite small. From our experience designing delay-sum beamformers, I/O pin limitations have generally been the main limitation on the amount of parallelism we can extract from a single FPGA, not the amount of FPGA logic.

Second, the FPGA system can access more memory every cycle, meaning that it can support more MAC operations simultaneously. Said another way, the FPGA's I/O can be tailored more easily to the application than the DSP. In this application, each FPGA is able to support larger amounts of memory and more memory ports than a SHARC. As mentioned above, the number of FPGA-based beamformers per FPGA is limited mainly by the number of user I/O pins of the FPGA in question since more I/O pins translates directly into more memory ports . More parallelism can be achieved at the cost of more expensive, higher I/O FPGAs.

Third, the FPGA-based beamformers with their application specific address generators are able to fetch all of the operands of interest in parallel each cycle, allowing the FPGA-based beamformers to complete a MAC every cycle. The SHARC address generation scheme, while great for other kinds of MAC operations (FIR filters and FFTs), is not flexible enough to perform the delay-sum beamforming MAC in a single cycle.

4.2.2 Price/Performance

As for the price/performance aspect of the FPGA-SHARC comparison, FPGAs' price-performance for beamforming can be comparable with that of DSPs if not much better. A 40-MHz ADSP-21060 DSP costs about $325 in small amounts. For the example beamforming application, the FPGA of choice is a Xilinx XC4028XL-1. At a cost of about $300 in small quantities, the FPGA enjoys about a 6 to 1 price/performance advantage.

For the spherical array example, the FPGA of choice is a Xilinx XC4052XL-1 which costs about $1000 in small quantities. Even with this cost, the FPGA still has about a 4 to 1 price/performance ratio advantage over the SHARC 21060 DSP.

4.3 Interprocessor Communication

Though one might expect the FPGA-based multicomputer to have more efficient interprocessor communications than the SHARC-based solution because of the FPGA's ability to specialize its I/O functions, we did not find any significant difference between the two implementation methods for the beamforming algorithm provided here. This is true mainly because very little interprocessor communication is actually required for our example algorithm in a beam-oriented organization. Also, considering the communication does occur, all but a little bit of it can be done transparently to the operation of the beam processing on both platforms.

The proposed FPGA system does have one disadvantage in this regard, though: the host processor cannot load the new data samples into the FPGA memories without disturbing the beam processing. Considering that this memory transfer requires only 400 cycles while the calculation of an entire set of beams require $20,000$ cycles, this is not a very large communication cost. If this ever became a large bottleneck, one effective but costly solution would be to make the memories dual-ported so the host can access the memories directly.

As a general comment, one potential problem with the SHARC DSP shared bus configuration is that the only way to add additional memory resources with a high-bandwidth data path to the SHARC is to attach the memory to the shared multiprocessor bus—leading to increased bus and memory contention. Several companies have found ways to isolate or decouple SHARCs and associated memories from the multiprocessor bus to increase the effective bandwidth between DSP and external memory and reduce bus and memory contention, but the various solutions can complicate the interprocessor communication model for SHARC clusters, especially since the host processor may be hindered from direct access to DSP internal memories. Clearly, with the flexibility of FPGAs' I/O, this problem with adding additional memory is not one generally encountered in properly designed FPGA-based systems.

4.4 Design Process

When considering multiprocessing designs, we find it hard not to mention the programming methods required. Needless to say, both high-performance DSP and FPGA multiprocessing systems are challenging to create and program.

Designers often use either schematic capture or an HDL to describe the computations performed by FPGAs. This process can be quite tedious and is performed at a very low level, meaning the development time can be long. To its benefit, though, hardware is generally organized and designed as a collection of several modules/circuits executing in parallel, so parallelism is naturally expressed in FPGA designs.

On the other hand, DSPs can be programmed at a much higher level, allowing for more designer productivity in general. But, eventhough DSPs can be programmed with C and other compilers, DSP performance results are at maximum when much of the work done is handled by hand-crafted assembly code, either in the form of user-created routines or in the form of library calls hand-crafted by system vendors. The design process is complicated even more when parallel applications must be designed. Frequently, the multiprocessing interface for a cluster of DSPs must be created for each application or new board—no specific multiprocessing framework may exist for the given DSP multiprocessing system. So, programming a multiprocessing DSP system is generally not as easy as programming some cluster of scientific workstations—there is not that kind of software support. In summary, multiprocessor DSP applications are created in software at much higher level than the FPGA designs, but they are still a challenging task to create, especially when compared to many general-purpose multiprocessing systems.

5 Future Work

To alleviate the difficulty of creating beamforming designs for FPGAs, we will be working on CAD tools which will provide ways of automatically generating beamforming processors from a high-level description—dataflow graphs and some form of programming language are both candidate description methods.

We will also be spending much of our time learning about how to mix DSPs, FPGA-based computers, and general-purpose processors (GPPs) into a heterogeneous multiprocessing system. The assumption is that all three have their strong and weak points and combining them in heterogeneous systems may have performance and other benefits.

Additionally, we want to quantify how the power consumption of clusters of FPGA-based computers, DSPs, and GPPs each compare. Can FPGAs have some benefit over their programmable counter-parts?

In addition to these studies, we intend to continue our research in mapping much more complex beamforming algorithms to FPGA-based systems. A comparison of these algorithms implemented on FPGA and DSP platforms will also be an important part of our future work since DSP platforms are generally the most common choice for digital sonar beamforming.

6 Conclusions

Because of the flexibility of FPGAs, their communications and functions can be specialized to provide higher performance than multi-DSP systems for some applications, such as the delay-sum beamforming designs discussed above. This is true even when the basic calculations can be done effectively by DSPs (e.g.,MACs). From our experience with delay-sum beamforming, we believe that FPGAs can be effectively used for parallel computing applications which are now typically implemented with multiple DSPs. One of the main problems with using FPGAs in multiprocessor beamforming systems is the lack of CAD and software support, an issue we will be addressing in our future work.

References

[1] P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 88–102, 1993.

[2] A. L. Abbott, P. M. Athanas, L. Chen, and R. L. Elliott. Finding lines and building pyramids with Splash 2. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–161, Napa, CA, April 1994.

[3] Laurent Moll, J. Vuillemin, and P. Boucard. High energy physics on DECPeRLe-1 programmable active memory. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 47–52, Monterey, CA, February 1995.

[4] N. Shirazi, P. M. Athanas, and A. L. Abbott. Implementation of a 2–D fast fourier transform on an FPGA–based custom computing machine. In W. Moore and W. Luk, editors, *Field-Programmable Logic and Applications. 5th International Workshop on Field-Programmable Logic and Applications*, pages 282–292, Oxford, UK, September 1995. Springer-Verlag.

[5] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, April 1996.

[6] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86–92, Monterey, CA, February 1997.

[7] P. Graham and B. Nelson. Genetic algorithms in software and in hardware — A performance analysis of workstation and custom computing machine implementations. In J. Arnold and K. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 216–225, Napa, CA, April 1996.

[8] Barry Van Veen and Kevin Buckley. Beamforming: A versatile approach to spatial filtering. *IEEE ASSP Magazine*, 5(2):4–24, April 1988.

[9] Don H. Johnson and Dan E. Dudgeon. *Array Signal Processing: Concepts and Techniques*. Prentice Hall Signal Processing Series. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[10] Norman L. Owsley. *Array Signal Processing*, chapter Sonar Array Processing, pages 115–193. Prentice-Hall Signal Processing Series. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[11] Nanette Boden, Danny Cohen, Robert Felderman, Alan Kulawik, Charles Seitz, Jakov Seizovic, and Wen-King Su. Myrinet—a gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

FPGA Implementation of the Pixel Purity Index Algorithm for Hyper-Spectral Images

LA-UR # 00-2466

March 2000

Dominique LAVENIER[1], James THEILER[2]

Los Alamos National Laboratory
[1]NIS-3: Space Data System
[2]NIS-2: Space and Remote Sensing Science
Los Alamos - NM 87545 - USA

Abstract

This technical report describes a hardware implementation of the Pixel Purity Index algorithm for hyper-spectral images. The realization has been experimented on a commercial FPGA board, the Wildforce board from Annapolis Micro System Inc. Speed-up of 80 has been achieved, reducing the computation time to several minutes compared to a few hours on standard sequential machines.

1 Introduction

The Pixel Purity Index (PPI) algorithm is part of a process for finding "end-members" in a hyper-spectral image, where the end-members can be defined as the spectral signature of this image. Basically a hyper-spectral pixel reflects the material composition of a given point of a particular scene, and the idea behind the notion of end-members is that a pixel is a linear combination of only a few distinct materials.

A geometrical interpretation of the linear combination in a D-dimensional Euclidean space is that the end-members are vertices of a convex polyhedron, and that the data are inside this polyhedron. The PPI algorithm attempts to identify the bounding D-dimensional polyhedron by finding vertices of the convex hull of data. These vertices correspond to particular data points.

The algorithm proceeds by generating a large number of random D-dimensional vectors, called skewers, through the hyper-spectral image. For each skewer, every data point is projected onto the skewer, and the position along the skewer is noted. The data points which correspond to extrema in the direction of a skewer are identified, and placed on a list. As more skewers are generated, this list grows. The number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the most pure, and a pixel's count provides its "pixel purity index".

The complexity of the PPI algorithm is in $O(K \times D \times N)$ with K the number of skewers, D the number of spectral bands and N the number of pixels. This is a very time consuming algorithm. As an example, processing a single 512×614 224-channel satellite image takes more than three hours on a 450 MHz processor (4K skewers). In that case, real-time analysis can only be achieved using specific architectures exploiting some interesting features of the PPI algorithm.

Fortunately, the PPI algorithm is a good candidate for hardware acceleration because it is readily parallelizable: all the computations perform on the K skewers are independent, leading to a high potential degree of parallelism, knowing that K is in the range of a few thousands!

The hardware implementation described in the next sections is heavily based of the ability to perform concurrently these computations onto FPGA components. The target reconfigurable system is the Wildforce accelerator board from Annapolis Micro System Inc [1], and the implementation we propose reduces the computation time to a few minutes compared to hours.

The next section presents the PPI algorithm and its parallelization. Section 3 gives the principle of the architecture and details the main components. Section 4 is devoted to the implementation on the Wildforce board.

2 Parallelization of the PPI Algorithm

The input data of the Pixel Purity Index algorithm are: (1) a hyper-spectral image composed of N pixels. Each pixel is a vector of D values of 16 bits; (2) a set of K random vectors (skewers) of size D. The output is an integer vector Q of size N.

As stated in the previous section, the key parameters of this algorithm are K, D and N which respectively represent the number of skewers, the number of spectral bands, and the number of pixels. The computation time essentially depends on these three parameters. The sequential C-like algorithm description is:

```
PIXELS[N][D];          // an image of N pixel vectors of size D
SKEWER[K][D];          // a set of K random vectors of size D (skewers)
Q[N];                  // the result array

for (n=0; n < N; n++) Q[n]=0;               // reset Q
for (k=0; k < K; k++)  {                    // for the K skewers
   for (n=0; n < N; n++) {                  // for the N pixels
      dp = 0;
      for (d=0; d < D; d++) {               // compute a Dot Product
         dp = dp + SKEWERS[k][d]*PIXELS[n][d];
       }
      if (dp > dpmax) { imax=n; dpmax=dp; }  // max computation
      if (dp < dpmin) { imin=n; dpmin=dp; }  // min computation
     }
   Q[imax]++;
   Q[imin]++;
}
```

For each skewer, N dot-products are computed in order to determine the two pixels which have produced the higher and the lower dot-product. The Q vector is modified accordingly.

As stated before, the $N \times K$ dot-products can be performed independently. If we supposed that:

1. the hardware can support concurrently the computation of P dot-products,
2. NS pixels can be accessed simultaneously,
3. KS skewers can be accessed simultaneously

such that $P = NS \times KS$ then we can re-express the algorithm as follows:

```
for (k=0; k < K; k=k+KS) {
   for (n=0; n < N; n=n+NS) {
      forall (ks=0, ns=0; ks<KS, ns<NS; ks++, ns++) {
         dp[ks][ns] = Dot-Product(SKEWERS[k+ks],PIXELS[n+ns]);
         if (dp[ks][ns] > dpmax[k+ks]) {
            imax[k+ks]=n+ns; dpmax[k+ks]=dp[ks][ns];
          }
         if (dp[ks][ns] < dpmin[k+ks]) {
            imin[k+ks]=n+ns; dpmin[k+ks]=dp[ks][ns];
          }
       }
    }
   for (ks=0; ks < KS; ks++) {
      Q[imax[k+ks]]++;
      Q[imin[k+ks]]++;
    }
 }
```

The `forall` loop computes concurrently P dot-products, each dot-product requiring D multiplication/accumulations.

3 Architecture

3.1 Principle

The principle of the architecture is represented figure 1. It is composed of a matrix of $NS \times KS$ dot-product operators. Each dot-product is fed serially with the D the pixels and the skewers. As the pixels and the skewers are D-vector data, D cycles are required for computing P dot-products.

The results of the dot-product are stored into registers (shaded boxes) and shifted to KS MinMax units. These units compute the minimum and the maximum of the dot-product and kept trace of the pixel numbers which have produced these extrema values. The results of the MinMax units are shifted to the host processor through a fifo.

In order to get maximum performances, the minimum and maximum operations are performed in parallel with the dot-product computation: As soon as a dot-product phase is accomplished, the

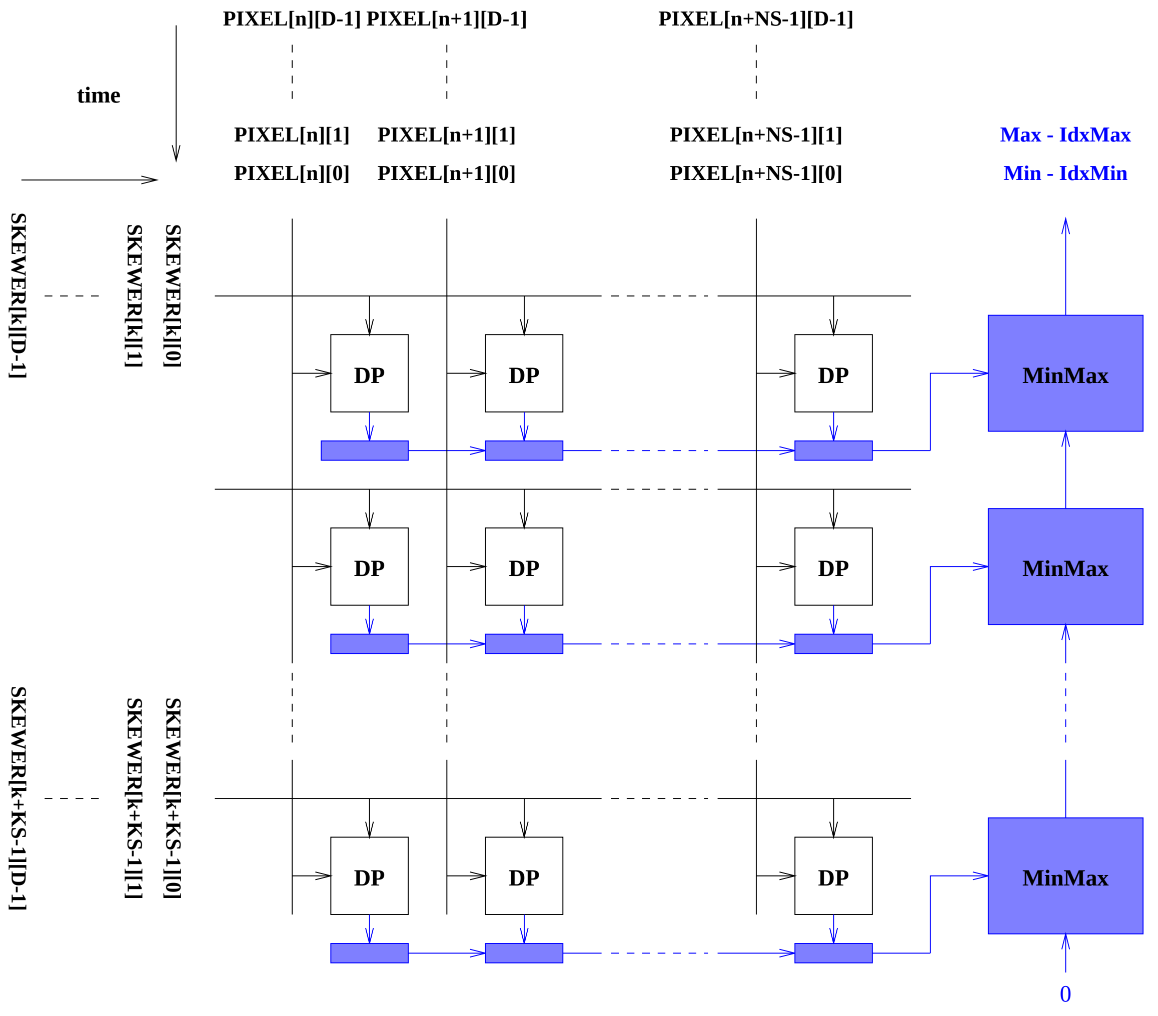


Figure 1: Architecture Principle

results are stored in the shift registers, and another phase begins immediately. During the next dot-product phase computation, the previous dot-product values are bit-serially shifted to the MinMax units. Hence, there is a complete overlap between the dot-product and the minimum/maximum computations.

Note that the updating of the Q array is not performed by the architecture. Since it represents a very small percentage of the overall computation, it is left to the host responsibility.

3.2 The Dot-Product Operator

The dot-product operator is an optimized 16-bit multiplier/accumulator. Pixels are 8-bit encoded (bits 13 to 5 from the initial 16-bit value) and skewers are 3-bit encoded. The skewer values range from -2 to +2. Consequently, the multiplication with a pixel by 0, 1 or 2 is straightforward: it is a null value, the pixel itself, or the pixel 1-bit left-shifted. The table below gives the skewer encoding:

Skewer Encoding			Corresponding
Skewer[2]	Skewer[1]	Skewer[0]	Decimal Value
0	0	0	0
0	0	1	-1
0	1	0	-2
0	1	1	x
1	0	0	x
1	0	1	1
1	1	0	2
1	1	1	x

Figure 2 details the architecture of the dot-product operator. The heart is an adder/subtractor-accumulator provided by the Xilinx logiblox library from the Alliance Series FPGA CAD tools [3]. Its input is connected to a multiplier unit which is actually reduced to a multiplexer gate for providing either a null value, a direct pixel or a 1-bit left-shifted pixel value. The `Reset` signal reset the accumulator to zero and load the shift register with the last computed dot-product.

The VHDL code of the Dot-Product operator is given in Annex A.

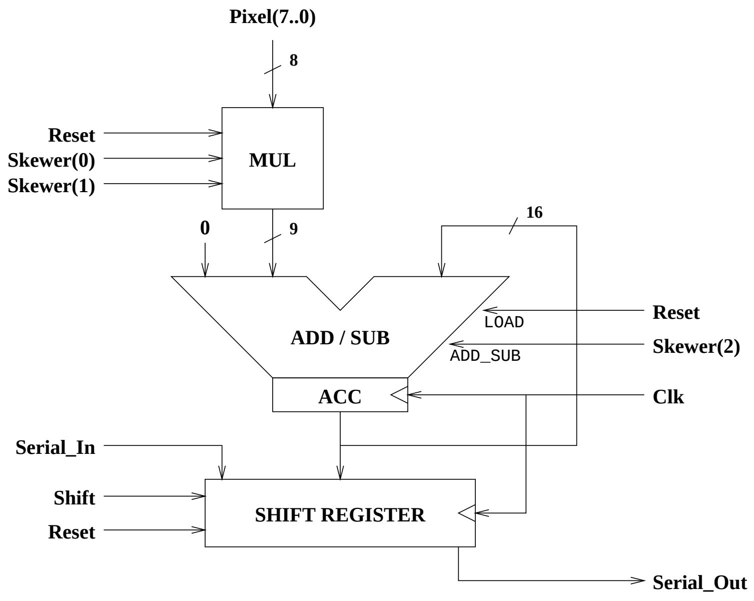


Figure 2: Dot-Product operator

3.3 The MinMax Unit

A MinMax unit receives bit-serially a stream of dot-product results. It has the charge of detecting the maximum and the minimum values together with the pixel index which has provided these extrema. It output serially two 32-bit data, first the maximum dot-product associated with its pixel index and, second the minimum dot-product and its associated pixel index. The lower 16 bits encode the min or max value and the upper 16 bits the pixel index. These 32-bit data are sent to the host.

The architecture is shown in figure 3. The `Buffer_in` register input serially 16 bits corresponding to a dot-product result. Thus, every 16 cycles, a comparison between the value stored in the `Buffer_in` register and the `AccMax` and `AccMin` registers is performed to update or not these registers.

After a complete cycle is achieved for computing the N dot-products against one particular skewer to every pixels, the maximum and minimum dot-products are sent to the host. The MinMax units are cascaded so that the host receives alternatively a maximum dot-product (with its pixel index) and a minimum dot-product (with its pixel index).

The input labeled `# Pixel` comes from a counter incremented each time a new dot-product is entered into the `Buffer_in` register. This counter is common to all the MinMax unit and computes the pixel index.

The VHDL code of a MinMax unit is given in annex B.

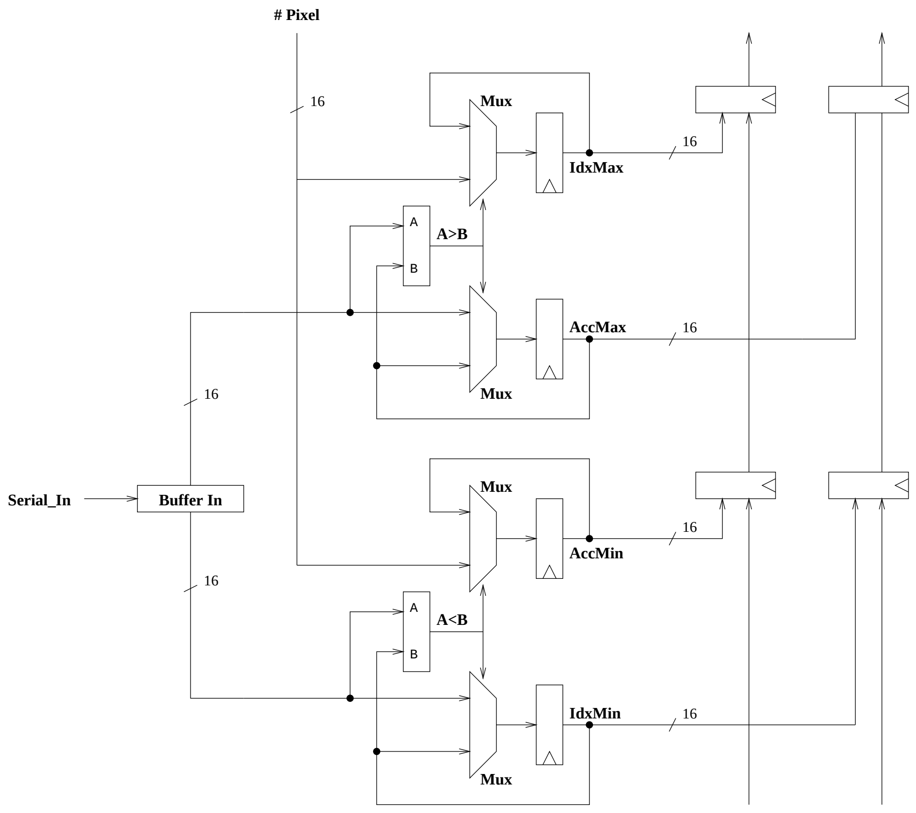


Figure 3: MinMax Unit

3.4 Control

There are two processes running concurrently. The first one controls the dot-product array (dot-product process), and the second one controls the MinMax units (minmax process). Both processes are implemented with a state machine as shown in Figure 4.

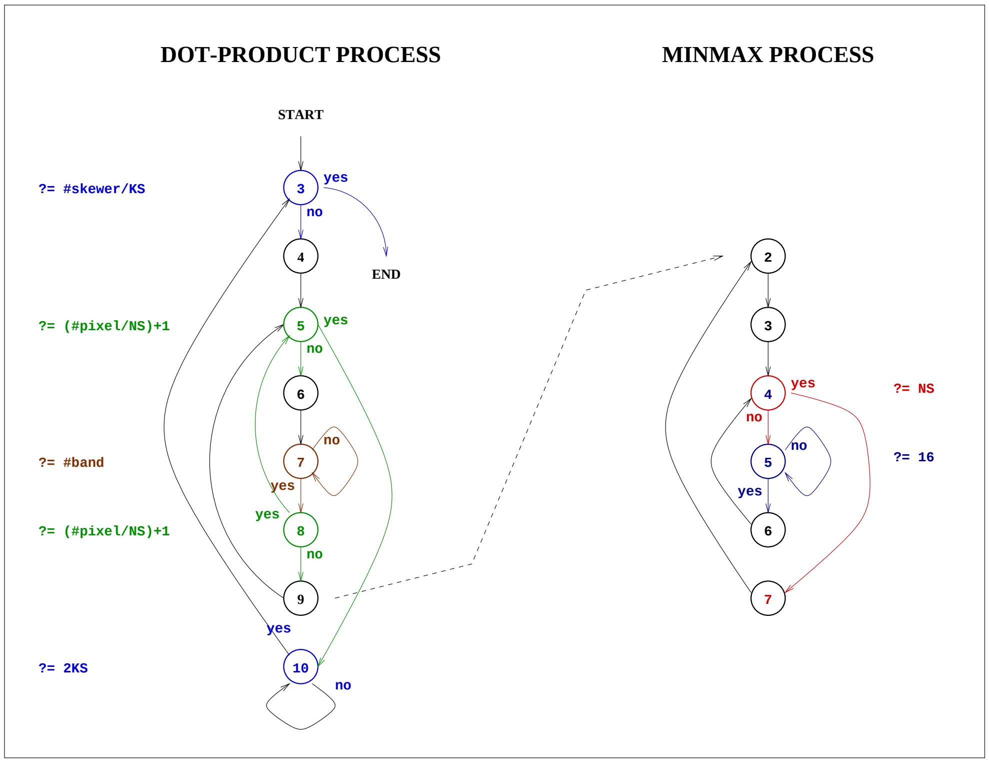


Figure 4: Control

The dot-product process is the main process. The outer loop performs (K/KS) iterations corresponding to the number of skewers (K) divided by the number of skewers concurrently computed (KS). Each iteration computes N dot-products (N = number of pixels). Knowing that NS pixels are processed in parallel, (N/NS) steps are required. Actually, and as it can be noticed on Figure 4, (N/NS)+1 steps are performed. This is due to the fact that the minmax computation of step i is achieved during the computation of step $(i+1)$. At the end of this step (except for the last one) the minmax process is activated (state 9 on figure 4).

The minmax process performs NS iterations. Each iteration first get serially KS different dot-products (16 cycles), then compute concurrently KS minimum and maximum according to these new dot-product values. To work properly, the time for computing a dot-product and the time for processing the results must respect the following constraint:

$$D >= (16 * NS)$$

The time for computing a dot-product (D cycles for a hyper-spectral image of D bands) must be greater (or at least equal) to the time for processing the min and max (precision of the results multiplied par the number of pixels processed concurrently).

At the end of each outer loop iteration, the dot-product process send KS results to the host. A result is composed of 2 pairs of (value,index) representing the maximum and minimum with their associated pixel index. Although further computation on the host to update the Q vector only required the index, the maximum and minimum are also provided. Actually, they are needed for partitioning problem if the image or the skewers cannot fit in the board memory.

3.5 Partitioning

Partitioning is required if the reconfigurable board cannot store a complete hyper-spectral image (N D-vector pixels) and the K skewers in its memory. In that case, the computation has to be performed into several passes.

Let us assume that only K_p skewers and N_p pixels can be fit simultaneously to the board memory. The best way to minimize data transfer between the host and the board is to apply the PPI algorithm on sub-hyper-spectral images of N_p pixels, and for each sub-image to have K/K_p successive passes.

The partitioned algorithm is:

```
for (n=0; n<N/Np; n=++)
 {
   load_pixel (n*Np,(n+1)*Np);    // load a sub-image of Nb pixels
   for (k=0; k<K/Kp; k=k++)
    {
      load_skewer (k*Kp,(k+1)*Kp; // load Kp skewers
      PPI (Np,Kp);                // process PPI
    }
 }
```

In that partitioning scheme, an overhead, compared to the non-partitioned algorithm is introduced by reloading several times the skewers. But, actually, this overhead is very small: the PPI algorithm is a computed-bound problem and data transfer have a little of influence over the total execution time.

4 Implementation on the Wildforce Board

The Wildforce board is marketed by Annapolis Micro System Inc [1]. The board available at LANL is mainly composed of five Xilinx XC4036EX processing elements [2] interconnected in a 36-bit width systolic ring. Each processing element, except CPE0, is connected to a 512 Kbytes memory (128 K 32-bit words). The processing elements CPE0, PE1 and PE4 are connected to the PCI interface bus by a bidirectional 512 32-bit word fifos. The memories are dual port memories accessible both from the host (thru the PCI bus) and the processing element. A programmable crossbar connects all the processing elements.

The architecture described in the previous section (figure 5) is split into the four processing elements PE1, PE2, PE3 and PE4. CPE0 is not used. PE2, PE3 and PE4 house each 32 dot-product operators, leading to a total of 96 operators able to run concurrently. PE1 contains 8 MinMax

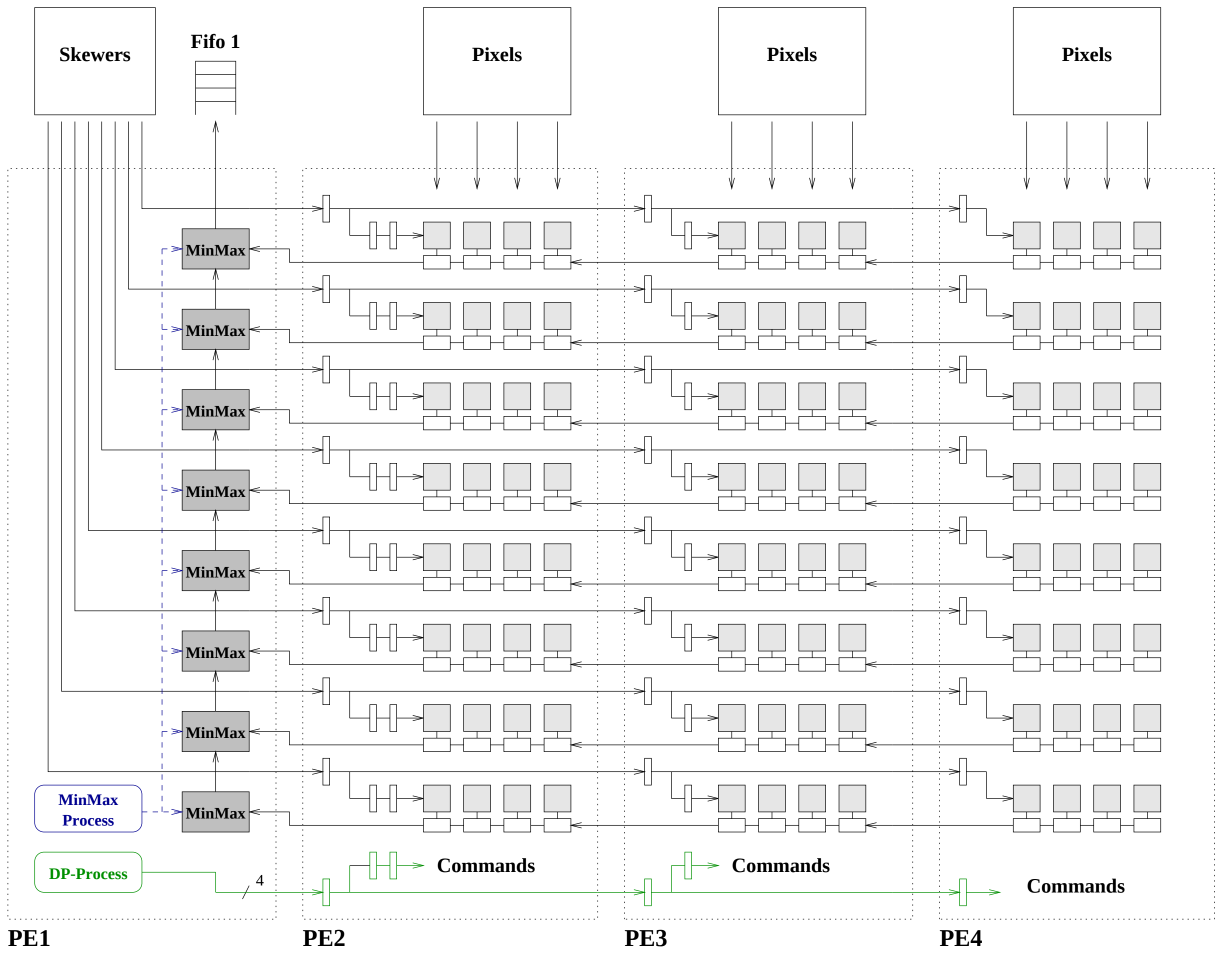
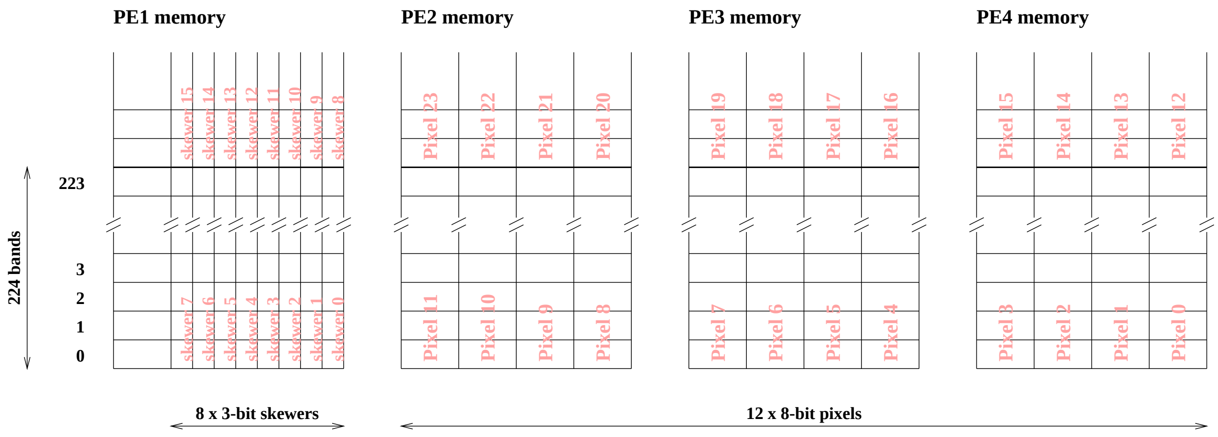


Figure 5: Wildforce Implementation

units and the two state machines. The memories of the processing elements PE2 to PE4 contain the pixels. The memory of PE1 contains the skewers.

The control located into the processing element PE1 is pipelined through the processing elements PE2 to PE4. The skewers, located into the memory of PE1, are propagated in the same way. The crossbar could have been used to broadcast the control and the skewers, avoiding this relatively complex control. There is one main reason for not using the crossbar: the internal PE pin connection does not fit with the layout organization of the architecture. The skewers have to flow "horizontally" while the crossbar in/out pins are provided "vertically". This results in long wires for connecting the skewers to the dot-product operators, leading to long delays and a low working frequency as experimented in some preliminary versions.

A sub hyper-spectral image is stored into the memory of the processing elements PE2 to PE4, while the memory of PE1 stores the skewers. The pixels and the skewers are stored as follows:



Since the size of a PE memory is 512 Kbytes, it can contain a maximum of 2340 224-band pixel. In other words, the board memory dedicated for storing a hyper-spectral image can hold a maximum of 7020 pixels. The current implementation stores 6144 pixels (512 $\times$ 12): each processor stores 2048 pixels and can simultaneously access 4 of them. Similarly, the PE1 memory stores 4096 skewers (512 $\times$ 8) and a single memory access provides 8 values.

The design has been described in VHDL, and synthesized with Synplify from Synplicity Inc. [4]. Manual placement has been performed to get a "regular" layout array in order to optimize the place-and-route step and obtain better performance. The design works at 25 MHz.

Performance

To determine the speed-up over a sequential programmable machine, the same algorithm has been coded in C and optimized accordingly. Tests have been made on real data provided by satellite remote sensors:

- image = 512 $\times$ 640 pixels
- 224 channels
- 4 000 skewers

A speed-up of 80 has been measured over a 450 MHz PC with 256 Mbytes SDRAM and running the LINUX system. In other words, a process taking 190 minutes can be reduced to 140 seconds (2 mn, 20 sec). The speed-up is calculated as the ration between the execution time (given by the unix command `time`) of the sequential algorithm and the wildforce implementation. Both include the time for accessing the data through the network.

References

[1] Wildforce Reference Manual, revision 3.4, Annapolis Micro System Inc, 1999 (www.annapmicro.com).

[2] Xilinx XC4000E and Xilinx XC4000X Series Field Programmable Gate Array, Product specification, version 1.6, Xilinx Inc, 1999 (www.xilinx.com).

[3] Alliance Series 2.1i, Xilinx Inc, 1999 (www.xilinx.com).

[4] Synplify User Guide, release 5.2.2, Synplicity Inc, 1999 (www.synplicity.com).

Annex A Dot-Product VHDL code

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mac2 is
 port
   (
     Clk       : in  std_logic;
     ClkEn     : in  std_logic;
     Reset     : in  std_logic;
     Shift     : in  std_logic;
     SerialIn  : in  std_logic;
     SerialOut : out std_logic;
     PixelIn   : in  std_logic_vector (7 downto 0);
     SkewerIn  : in  std_logic_vector (2 downto 0)
   );

end mac2;

architecture struct of mac2 is

component addsubacc
  PORT(
    ADD_SUB: IN std_logic;
    B      : IN std_logic_vector(15 DOWNTO 0);
    LOAD   : IN std_logic;
    CLK_EN : IN std_logic;
    CLOCK  : IN std_logic;
    Q_OUT  : OUT std_logic_vector(15 DOWNTO 0));
end component;

component mul_pix is
  port (
         SK0R : in std_logic;
         SK1  : in std_logic;
         Pi   : in std_logic;
         Px   : in std_logic;
         M    : out std_logic);
end component;

component Sk0Reset is
  port (
         Sk0  : in std_logic;
         Sk1  : in std_logic;
         Raz  : in std_logic;
         SK0R : out std_logic);
end component;
```

```
  signal Acc   : std_logic_vector (15 downto 0);
  signal Mul   : std_logic_vector (15 downto 0);
  signal Srg   : std_logic_vector (15 downto 0);
  signal SK0R  : std_logic;
  signal Zero  : std_logic;

begin

Zero <= '0';
SerialOut <= Srg(15);

asa0 : addsubacc port map
(ADD_SUB => SkewerIn(2),
 B       => Mul,
 LOAD    => Reset,
 CLK_EN  => ClkEn,
 CLOCK   => Clk,
 Q_OUT   => Acc);

skk: Sk0Reset port map (SkewerIn(0),SkewerIn(1),Reset,SK0R);
mp0: mul_pix port map (SK0R, SkewerIn(1),PixelIn(0),Zero,      Mul(0));
mp1: mul_pix port map (SK0R, SkewerIn(1),PixelIn(1),PixelIn(0),Mul(1));
mp2: mul_pix port map (SK0R, SkewerIn(1),PixelIn(2),PixelIn(1),Mul(2));
mp3: mul_pix port map (SK0R, SkewerIn(1),PixelIn(3),PixelIn(2),Mul(3));
mp4: mul_pix port map (SK0R, SkewerIn(1),PixelIn(4),PixelIn(3),Mul(4));
mp5: mul_pix port map (SK0R, SkewerIn(1),PixelIn(5),PixelIn(4),Mul(5));
mp6: mul_pix port map (SK0R, SkewerIn(1),PixelIn(6),PixelIn(5),Mul(6));
mp7: mul_pix port map (SK0R, SkewerIn(1),PixelIn(7),PixelIn(6),Mul(7));
mp8: mul_pix port map (SK0R, SkewerIn(1),Zero,      PixelIn(7),Mul(8));
Mul(15 downto 9) <= "0000000";

process (Clk)
  begin
    if rising_edge(Clk) then
      for i in 15 downto 1 loop
 Srg(i) <=    ( (Reset    ) and Acc(i) )
                   or ( (not Reset) and
                                       (    ( (Shift    ) and Srg(i-1) )
                                         or ( (not Shift) and Srg(i)   )
                                       )
                      );
 Srg(0) <=    ( (Reset    ) and Acc(0) )
                   or ( (not Reset) and
                                       (    ( (Shift    ) and SerialIn )
                                         or ( (not Shift) and Srg(0)   )
                                       )
                      );
      end loop;
    end if;
  end process;

end struct;
```

Annex B MinMax VHDL code

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity minmax is
 port
   (
     Clk      : in  std_logic;
     RazAcc   : in  std_logic;
     LoadAcc  : in  std_logic;
     LoadOut  : in  std_logic;
     ShiftIn  : in  std_logic;
     ShiftOut : in  std_logic;
     SerialIn : in  std_logic;
     IdxPix   : in  std_logic_vector (15  downto 0);
     DataIn   : in  std_logic_vector (31 downto 0);
     DataOut  : out std_logic_vector (31 downto 0)
   );
end minmax;

architecture struct of minmax is

component AinfB16
  PORT(
    A: IN std_logic_vector(15 DOWNTO 0);
    B: IN std_logic_vector(15 DOWNTO 0);
    A_LT_B: OUT std_logic);
end component;

component AsupB16
  PORT(
    A: IN std_logic_vector(15 DOWNTO 0);
    B: IN std_logic_vector(15 DOWNTO 0);
    A_GT_B: OUT std_logic);
end component;

  signal AccMin    : std_logic_vector (15 downto 0);
  signal AccMax    : std_logic_vector (15 downto 0);
  signal OutMin    : std_logic_vector (15 downto 0);
  signal OutMax    : std_logic_vector (15 downto 0);
  signal IdxAccMin : std_logic_vector (15 downto 0);
  signal IdxAccMax : std_logic_vector (15 downto 0);
  signal IdxOutMin : std_logic_vector (15 downto 0);
  signal IdxOutMax : std_logic_vector (15 downto 0);
  signal BuffIn    : std_logic_vector (15 downto 0);
  signal AsupB     : std_logic;
  signal AinfB     : std_logic;
```

```
begin

max : AsupB16 port map (BuffIn, AccMax, AsupB);
min : AinfB16 port map (BuffIn, AccMin, AinfB);

DataOut (31 downto 16) <= IdxOutMax;
DataOut (15 downto  0) <= OutMax;

process (Clk)
begin
  if rising_edge (Clk) then
            if RazAcc = '1' then
      AccMax <= (others => '0');
      AccMin <= (others => '1');
      Buffin <= (others => '0');
            else
      if ShiftIn = '1' then
        for i in 15 downto 1 loop
          BuffIn(i) <= BuffIn(i-1);
                end loop;
        BuffIn(0) <= SerialIn;
              end if;
      if LoadAcc = '1' then
if AsupB = '1' then
                  IdxAccMax <= IdxPix;
                  AccMax <= BuffIn;
                end if;
if AinfB = '1' then
                  IdxAccMin <= IdxPix;
                  AccMin <= BuffIn;
                end if;
      end if;
      if LoadOut = '1'  then
                OutMin <= AccMin;
        OutMax <= AccMax;
                IdxOutMin <= IdxAccMin;
        IdxOutMax <= IdxAccMax;
              end if;
      if ShiftOut = '1' then
        OutMax <= OutMin;
        IdxOutMax <= IdxOutMin;
                OutMin <= DataIn(15 downto 0);
                IdxOutMin <= DataIn(31 downto 16);
              end if;
    end if;
          end if;
        end process;

end struct;
```

FPGAs and DSPs for Sonar Processing — Inner Loop Computations*

Paul Graham and Brent Nelson
Brigham Young University
grahamp@ee.byu.edu, nelson@ee.byu.edu

Abstract

In this paper we evaluate FPGA technology for use in sonar processing, specifically, time-delay, frequency-domain, and matched-field beamforming algorithms. We compare our results with those obtained using DSPs to highlight a number of FPGA features which make them attractive for signal processing. First, the CORDIC algorithm maps well onto FPGAs, in our case allowing complex arithmetic to be done in either rectangular or polar forms. This greatly reduces the hardware required. Second, the ability of FPGAs to support multiple memory ports eliminates memory bottlenecks and allows multiple processing elements to be placed into each FPGA, increasing performance via parallelism. Finally, custom computational units and pipelining make possible single-cycle inner loop computations on FPGAs where DSPs require multiple cycles per inner loop. Based on these comparisons, FPGAs appear to be a reasonable alternative to DSPs for these algorithms.

1 Introduction

Many researchers have explored the use of configurable computing machines (CCMs) based on field-programmable gate arrays (FPGAs) in accelerating computation intensive algorithms. Custom computing machines are attractive since they employ application-specific hardware to accelerate the application while maintaining some degree of programmability and, thus, flexibility. In [1] the authors summarize a number of characteristics of image processing computations which make them suitable for FPGA-based implementation. These characteristics include: (a) high levels of parallelism exploitable by FPGAs, (b) simple operations performed in a fixed sequence amenable to pipelining, (c) low-resolution data (8-12 bits), (d) large data sets, and (e) simple control (or none at all). Our own informal survey of articles from the ACM FPGA and IEEE FCCM conferences reveals that a majority of published algorithms on FPGA systems meet these criteria.

*Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

For over the past year we have been working in the area of sonar beamforming [2]. In many ways sonar processing fits the criteria above — the computations are data-parallel, they require little control, the data sets are large (infinite streams), and the raw sensor data is at most 12-bits. However, they have three characteristics which make them challenging. First, they involve intensive arithmetic (multiply-accumulates and trigonometric functions) on real and/or complex data. Second, they require significant memory support, far beyond that indicated in much previously published work. Third, the scale of the computation is large, requiring (possibly) hundreds of FPGAs and high-bandwidth interconnections to meet real-time constraints.

Thus, in our research we have concentrated on large, distributed, FPGA-based CCM architectures. These are similar to and compatible with the two-level multicomputer architecture described in [3, 4]. We have applied this to a range of sonar processing techniques including time-domain beamforming, frequency-domain beamforming, and matched field processing. The core computation for these techniques is a multiply-accumulate — the first using real data the other two using complex data. Each of these algorithms presents unique challenges. In the first and third, available memory is the limiting factor. In the second and third, complex arithmetic presents special difficulties.

For each of these algorithms, we have done FPGA mappings and, in the case of the two beamformers, compared the results with DSPs (an obvious alternative technology). We have found that, somewhat surprisingly, FPGAs not only perform the inner loop calculations for beamforming efficiently but, for the algorithms investigated, consistently outperform similar multiprocessing DSP systems. This is in spite of the significant differences between the algorithms. In the end, a variety of FPGA characteristics contributed to the results. These include flexible I/O and memory interface capabilities, the use of fine-grain pipelining to reduce inner loop computations to a single cycle, and especially the use of CORDIC to simplify arithmetic requirements.

In the balance of this paper we first describe a two-level multicomputer having both FPGA and DSP nodes which we have targeted for our studies. Following this, we discuss the sonar applications we have mapped. We then compare our FPGA-only and FPGA-DSP implementations with those possible with both the SHARC and TMS320C6201 DSPs. After summarizing the features of both FPGAs and DSPs pertinent to our results we finish up by discussing how FPGAs may be usefully employed for digital signal processing based on these results.

2 A Two-level Multicomputer

With a desire to work on large computing problems, we have been helping to develop a CCM system for a variety of applications requiring large amounts of computational power [1]. The computing model chosen is the two-level multicomputer [3, 4] shown in Figure 1.

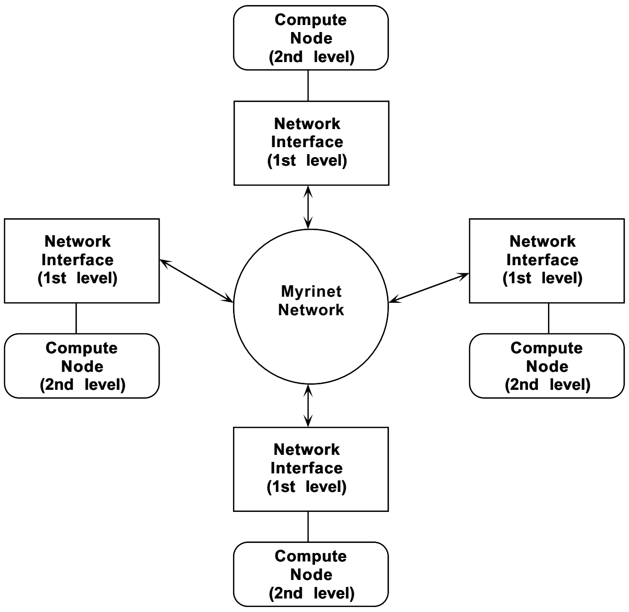

Compute Node (2nd level)
Network Interface (1st level)
Network Interface (1st level)
Network Interface (1st level)
Myrinet Network
Compute Node (2nd level)
Compute Node (2nd level)
Network Interface (1st level)
Compute Node (2nd level)

Figure 1: A Two-Level Multicomputer

In this organization, the first level of processors are connected through a high-speed network called Myrinet[5]. These specialized processors handle all communications, providing networking and other services to the second-level processors. The second-level processors execute user computations.

One advantage beyond the tremendous bandwidth of the Myrinet network (one 1.28 Gbit/sec input channel and one 1.28 Gbit/sec output channel per node) is that the second-level processors do not need complex control in the form of an operating system or runtime environment to operate effectively. This means that processors based on DSPs or FPGAs are likely candidates for second-level computing elements.

The proposed FPGA computing board for use in this multicomputer organization is shown in Figure 2. The board includes one Myrinet interface processor (a first-level node), one "host" FPGA, and four processing FPGAs (second-level nodes). The four processing FPGAs are connected in a simple ring and each has four 256Kx16 SRAMs attached to it. Though not shown, the host FPGA also has the same amount of memory as the processor FPGAs and the board has additional I/O connections to allow the FPGAs to connect directly to neighboring computing boards or other digital equipment.

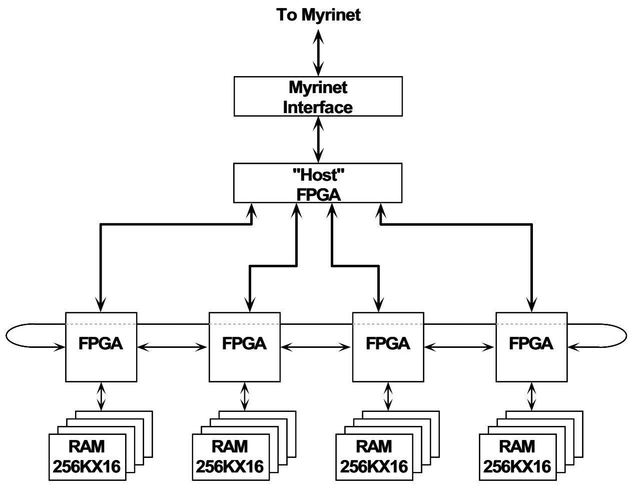


Figure 2: FPGA Multicomputer Board

3 FPGA-Based Sonar Beamforming

Beamforming is a spatial filtering operation performed on data received by an array of sensors, such as antennas, microphones, or hydrophones. It provides a system with the ability to "listen" directionally even when the individual sensors in the array are omnidirectional. Beamforming not only causes the system to be more sensitive to signals coming from a specific direction, but also attenuates noise

[1]We are participating in the DARPA-sponsored SLAAC project, consisting of a number of academic and government researchers. The SLAAC project is focused on developing a system like the one outlined here for use in a variety of high-performance applications.

and interference coming from other directions. The central idea in beamforming is to sum the signals received by an array of sensors such that signals arriving from a given direction are added coherently; this results in a maximum signal response in the desired direction, while signals arriving from other directions will not be added in-phase, meaning that they will experience destructive interference. Using a knowledge of the receiving array's geometry, the speed of sound in water (for sonar), and the desired direction of arrival, the designer can calculate the relative amount of phase shift a signal experiences as it propagates across the array and can thus compensate for the observed phase shift at each sensor to perform coherent summing of the signals.

Two major methods for doing beamforming are in common use. In [2], we discussed the FPGA implementation of a *time-domain* or *delay-sum* beamformer. In this design, the signals received at each sensor are simply delayed so that signals coming from the direction of interest all appear in phase. A second method, *frequency-domain* beamforming, performs the desired phase shifts by directly manipulating the phase component of the signals' Fourier domain representations. For a detailed treatment of beamforming and its many forms, we refer the reader to [6] and [7].

In this paper we assume a towed linear sonar array with 400 sensors which are sampled at 2 kHz. From this sensor data we form 10,000 beams. This example problem is based on an existing sonar array with which we are familiar.

3.1 Time-Domain Beamforming

Time-domain beamforming is performed by delaying the signal received by each sensor so that all signals coming from a given beam direction are in-phase. The pseudocode given in Algorithm 1 represents the delay-sum beamforming calculation for a single beam direction. Key observations regarding this calculation include:

Algorithm 1 Pseudo-Code for Time-Domain Beamforming (Single Beam)

```
formBeam(b) {
  response = 0;
  for (s=0;s<numSensors;s++)
    response += shade[b][s] *
                dataSamples[s][delayFunction(b,s)];
}
```

- The core computation is a dot-product between an array of coefficients (the *shade[]* array) and sensor data.
- The *dataSamples[]* array is a fixed-size buffer which holds a running history of the last N samples received by each sensor.
- The *delayFunction()* returns the index of the sample to use for the current sensor and beam.

Thus, the calculation is not a simple dot-product. Rather, the evaluation of *delayFunction()* using a memory lookup is required[2]. In the end, a very irregular memory access pattern into the *dataSamples[]* buffer emerges.

Mapping the calculation to an FPGA results in the processing element (PE) with a MAC unit, some control circuitry, and three memories. Two such PEs fit in a Xilinx 4028XL and operate at a maximum clock rate of just over 50 *MHz* using 12-bit sensor data, 10-bit delay values, and 6-bit shade values. As discussed in [2], this beamformer design is memory port limited. Four external memories support two PEs per FPGA. However, even for this relatively small FPGA, only about 60% of the FPGA's logic is utilized in this design.

The computation rate required is $2000\,Hz \times 400\,sensors \times 10,000\,beams = 8\,billion$ MACs per second. At 40 *MHz* each PE calculates $40\,million$ MACs per second and so at least 25 of the boards from Figure 2 are required. As discussed in [2], the irregular memory access pattern in this computation forces a SHARC DSP to spend three cycles per inner loop, the result being a single FPGA can perform the work of six SHARC DSPs for this computation.

3.2 CORDIC Computations and Frequency-Domain Beamforming

Frequency-domain beamforming does the phase-shifting and summing of the data in the frequency, or Fourier, domain. Figure 3 provides a block diagram of the frequency-domain computation. In the first stage of processing 256-point FFTs are performed on each sensor channel's data. In the second stage beams are formed by multiplying this block of complex data by a similarly sized array of weights to perform the phase-shift. Summing across sensor channels for each frequency bin, a 256-element column vector results. In the third stage an inverse FFT (IFFT) is performed on this vector to provide the time-domain beam response. Since the second and third stages are performed $10,000$ times to form $10,000$ beams, they represent the bulk of the computational load. For this computation, we assume the data values produced by the FFT stage are 32-bits each (16 magnitude and 16 phase).

[2]In the presence of curved wavefronts from near-field sources, computing *delayFunction()* on the fly would be extremely expensive and make it necessary to recompile the design for a new set of beam directions. As a result, *delayFunction()* is evaluated via a table-lookup into memory.

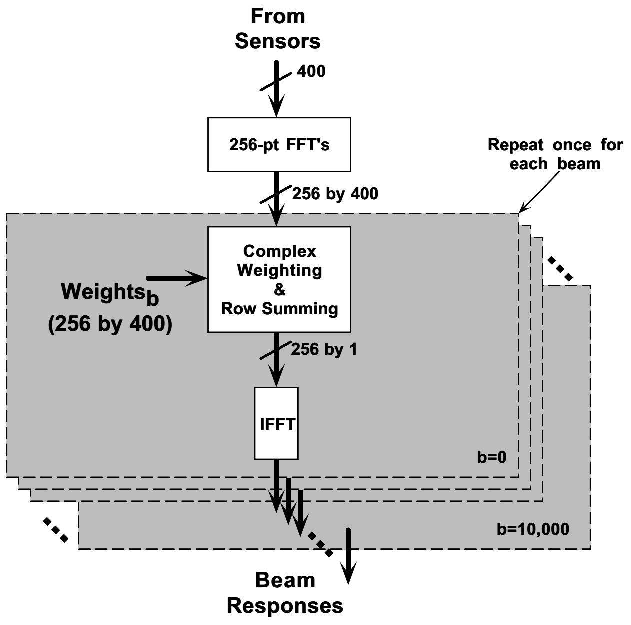


Figure 3: Frequency Domain Beamforming Algorithm

Algorithm 2 Pseudo-Code for Frequency-Domain Beamforming (single beam, 2nd stage of processing)

```
formBeam(b) {
  for (f=0;f<256;f++)
    response[f] = 0;
  for (s=0;s<numSensors;s++)
    for (f=0;f<256;f++)
      response[f] += weight[b][s][f] *
                     freqData[s][f];
  result = IFFT(response);
}
```

This algorithm, illustrated in Algorithm 2 has a number of characteristics that differentiate it from time-delay beamforming:

- In time-delay beamforming every PE operates on different data (all delays into the sample history are different). Here, each set of FFT outputs are repeatedly processed by all PEs to form the required number of beams.
- In this algorithm complex (real/imaginary) data and arithmetic computations are required.

Due to the ease and precision of performing FFT computations on a DSP, our implementation uses a single DSP board to compute all the FFTs and IFFTs shown in Figure 3. Thus, this discussion concentrates on performing the compute-intensive middle stage. An attractive implementation is to broadcast the FFT bin data to the SIMD array's host FPGAs, which repeatedly broadcast the data to the processing FPGAs, thus, reducing the number of memory ports on each PE. This is shown in Figure 4. With a single

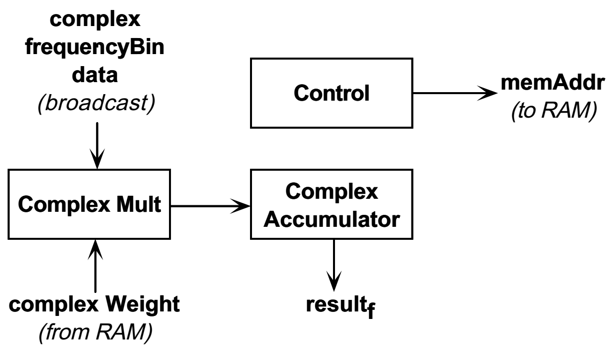


Figure 4: PE Computation for a Single Frequency Bin - Version 1

memory access per inner loop, up to four PEs might fit in an FPGA. However, while the computation looks simple, a complex multiply/accumulate requires four multiplies and four additions, expensive using FPGA technology[8].

The basic operation being performed here is simply a phase shift. An alternative is to perform the computation in polar, or phase-magnitude, form. This simplifies the phase shift (most easily done in polar form) but complicates the summation (most easily done in rectangular form).

The CORDIC algorithm is well-suited to conversion between these two formats. Before showing how CORDIC can reduce the computation required we now review basic CORDIC principles.

3.2.1 CORDIC Calculations

The CORDIC algorithm was introduced by Volder in the 1950's[9] and extended and unified by Walther in 1971[10]. Originally developed to compute basic trigonometric functions it has found wide application in a number of areas including the evaluation of the Fourier and Discrete Cosine transforms, solving linear systems, and eigenvalue problems to name but a few. As outlined in [11], these computations require the evaluation of elementary functions such as trigonometric, exponential, and logarithmic functions. A key reason for renewed interest in CORDIC in the context of VLSI and FPGAs is that these functions are difficult to compute efficiently using multiply-accumulate (MAC) hardware such as is found in DSPs or via other methods in VLSI.

The CORDIC algorithm to do a polar-to-rectangular conversion is a *rotation* and is described iteratively as:

$$\begin{aligned} x_0 &= polarMagnitude \\ y_0 &= 0 \end{aligned}$$

$$
\begin{aligned}
z_0 &= polarPhase \\
x_{i+1} &= x_i + \sigma_i \times 2^{-i} \times y_i \\
y_{i+1} &= y_i - \sigma_i \times 2^{-i} \times x_i \\
z_{i+1} &= z_i + \sigma_i \times \alpha_i
\end{aligned}
$$

where $\sigma_i = sign(z_i)$ and $tan(\alpha) = \pm 2^{-k}$. The key concept is that by choosing values of α's as indicated (their tangents are powers of 2), the entire computation consists of shift and add operations.

The above algorithm can be computed iteratively but for high-throughput single-cycle inner loop computations such as are found in beamforming it is advantageous to *unroll* the iteration and place pipeline latches between each iteration stage (this is called an *on-line CORDIC processor* in the literature). This eliminates the expensive shifter hardware (all shifts between stages are hard-wired). The result is that a fully pipelined 16-bit CORDIC unit occupies about the same area as a fully pipelined 16-by-16 multiply circuit on a XILINX 4000XL-series FPGA[3] and can be clocked at 40 *MHz*. In addition, the CORDIC FPGA layout is extremely regular. Andraka, in [12], does an excellent job of summarizing the CORDIC algorithm as does Hu in [11] — the reader is referred to these for more details on CORDIC computations.

Using CORDIC as outlined above, the frequency shift can be done in polar coordinates by a simple addition to the phase term[4]. This reduces the phase shift computation from four multiplies and two adds to one add and one CORDIC conversion[5], a significant savings considering that the multipliers account for most of the PE's size.

Finally, one additional enhancement must be made to the design to reduce the memory required to a feasible amount. Each weight has the form of a complex exponential: $e^{j \times \omega}$ which can also be expressed as $e^{j \times f \times \Delta t}$. For a given beam, the same Δt is used for all frequency bins associated with a single sensor. The computation is thus more efficiently done as shown in Figure 5. At the cost of an additional multiply, the memory required *per beam* is reduced from 200 KB (102,400 phase shift values) to about 800 bytes, a factor of 256 in memory savings. With this enhancement the construction of this beamformer is possible. A bonus is that the computational cost of the additional multiply is hidden via pipelining, something likely not possible in a programmable processor.

There are $b \times f \times s = 10,000 \times 256 \times 400 = 1.024\ billon$

[3]425 Xilinx 4000EX/XL CLBs for the CORDIC vs. 400 CLBs for the multiplier)

[4]This assumes the weight magnitude is unity.

[5]To take advantage of this savings, the first stage FFT processor will be required to convert its complex outputs from rectangular to polar format. Since this is done once for each window of data (i.e., each set of FFT outputs) and 10,000 beams are formed on this data, the cost of the conversion is insignificant.

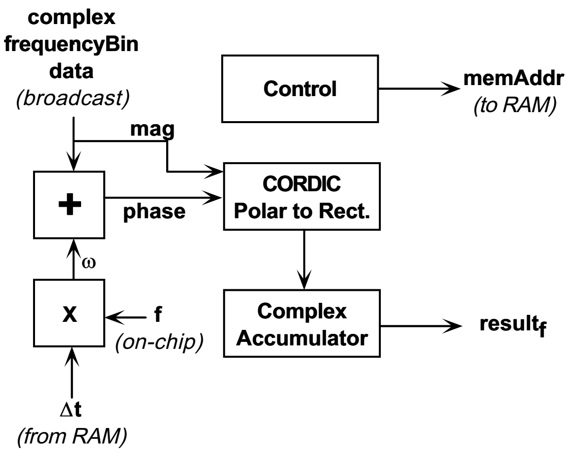


Figure 5: Magnitude-Phase PE Computation for a Single Frequency Bin - Version 2

phase shifts required for each group of FFT bins computed for a total of 8 billion phase shifts, 8 billion complex additions, and 8 billion CORDIC conversions per second. At 40 *MHz* per PE and two PEs per FPGA, 200 PEs or 25 boards are required for this part of the computation (the same as for the time-domain computation). Each such PE computes 50 beams and requires only about 40KB of memory to do so.

This PE was coded and simulated in VHDL and then synthesized to both Xilinx 4062XL and 4085XL parts, achieving a clock rate of 40 *MHz* for two PEs per FPGA. Due to the reduced memory requirements of this algorithm (it requires only a single external memory lookup per inner loop computation), the board design of Figure 2 will support four PEs per FPGA. This should be possible using larger Xilinx parts reducing the board count to half that of the time-domain case.

3.2.2 Matched Field Sonar Processing

A modified version of frequency-domain beamforming, which further leverages the advantages of CORDIC, is *matched field* sonar processing. As above, the computation progresses in 3 major stages (FFTs in stage one, phase shifts and summations in stage two, IFFTs in stage three). The unique feature of matched field processing is that the choice of the weight to use for each phase shift is a function of not only the beam number but also the target range (to the individual sensor)(r), the target's depth(Z_s), and the individual sensor's depth(Z_r).

This presents two problems. First, the number of weights that must be computed and stored is very large — 2.6GB worth for a typical problem, 90MB of which is in use at any given time. The storage, communication, and management of this data is problematic. Second, since the weights change regularly (the ocean environment changes, the ship or submarine maneuvers, ...), a method for updating the weights must also be provided. These weights may need to be computed several times per hour. Based on [13], the formula for this computation is as follows:

$$\begin{aligned} P(f,r,Z_s,Z_r) &= \frac{i}{\rho(Z_s)\sqrt{8\pi r}} \times e^{-\frac{i\pi}{4}} \times \\ & \sum_{m=1}^{M} \left[\frac{\phi(f,m,Z_r) \times \phi(f,m,Z_s)}{\sqrt{k(m,f)}} \times \right. \\ & \left. \frac{e^{i \times k(m,f) \times r}}{e^{\delta(m,f)) \times r}} \right] \end{aligned}$$

where M is a wave number and $\phi[]$, $k[]$, and $\delta[]$ all represent values which we can assume are computed elsewhere. The question may be asked whether computing the weights, storing, and distributing them to the beamformer processors is preferable to computing them as needed *on the fly*.

The straightforward computation of the weights in real-time using the equation above is prohibitive. However, two methods may be used to reduce the cost to a manageable level. First, unrolling the loop body (performing all loop iterations in parallel), reduces the f and M values to constants and eliminates a number of memory lookups. Second, as above, the weight computation can be done by computing the phase and magnitude parts of the summation body separately, converting to rectangular coordinates using CORDIC, performing the summation, and converting back to polar format for the remainder of the computation and for use downstream by the actual beamformer.

Each unrolled loop body requires only two memory accesses, one multiply, one constant-operand multiply, and one CORDIC rotation. Our current work indicates that with $M = 20$ and using the board architecture in Figure 2, it is likely cheaper to perform the unrolled computation and thus compute the weights as needed *on the fly* than to compute the weights once every 15 minutes and distribute them to the regular beamforming processors. The flexibility of FPGAs to support this computation has an additional advantage: it allows for more timely weight updates (as often as the $\phi[]$, $k[]$, and $\delta[]$ tables change).

3.3 Summary of FPGA–based Sonar Algorithms Implementations

In summary, FPGA flexibilities in a number of areas contribute to the results achieved for each implementation. For time-domain processing, the key to fitting multiple PEs into an FPGA is providing enough *independent* ports to external memory. FPGAs are not limited by a single memory interface as are programmable processors and thus additional I/O pins would allow more PEs per FPGA.

The frequency-domain beamformer benefits from a number of FPGA capabilities including:

- Simple synchronization with an external data source allows the broadcast of sensor data to all PEs in parallel, reducing the external memory accesses to one per inner loop.
- The efficient FPGA-based pipelined CORDIC unit makes it possible to freely mix polar and rectangular format computations.
- It was possible to reduce the memory requirements by a factor of 256 via the addition of an extra multiply in the inner-loop computation. This extra operation added no throughput penalty to the design since it was hidden via pipelining.

Similarly, the matched-field beamformer benefits from the use of CORDIC, both for the actual beamforming as well as for computing the required complex weights *on the fly*, resulting in a more adaptive algorithm. Heavy use of fine-grain pipelining further allows single-cycle execution of the weight computation.

4 Comparison of FPGA and DSP Approaches

In proposing that FPGA-based custom computing machines can be used for both time-domain and frequency-domain beamforming, we should compare our CCM implementations with other competing technologies. Considering that DSPs have been designed for performing digital filtering in the time- and frequency-domains, their comparison with FPGA-based implementations of the same algorithms would be useful. For our brief comparison, we will use two DSPs: the Analog Devices SHARC DSP and the soon-to-be-available Texas Instruments TMS320C6201.

We chose the SHARC DSP since it is a high-end DSP designed to support multiprocessing—a feature important to this problem. As a high-end DSP, the 40-MHz SHARC ADSP21060 performs a peak computation rate of 120 MFLOPs with a sustainable maximum computation rate of 80 MFLOPs and includes one of the largest on-chip memories for a DSP, 512 KB. The on-chip RAM is dual ported so that it can be accessed by the DSP's I/O devices and external hosts without disturbing the operation of the DSP's core, a feature which is clearly important if communication and computation are to be done simultaneously. Despite the fact that the SHARC is a floating-point DSP, the comparisons below will be concerned only with its fixed-point performance, treating the SHARC as a high-end, multiprocessing-savvy, fixed-point DSP. With peak and sustainable fixed-point computation rates equivalent to its floating-point rates, the SHARC is an excellent integer DSP.

We will also compare our FPGA implementations with a new fixed-point DSP from Texas Instruments, the TMS320C6201[6]. This DSP is different from all available DSPs in that it is a VLIW DSP. It has two load/store units which can double as 32-bit adders, two integer ALUs which can also perform shift operations, two additional integer ALUs, and two 16x16 multiplier units. Further, the DSP has 128 KB of on-chip memory, 64 KB for data and 64 KB for instructions. In contrast, the SHARC has 256 KB each for data and instructions with the ability to use the instruction memory for data storage also. This DSP also is distinguished by its clock rate — 200 MHz as compared to the typical DSPs which run at 20 MHz to 100 MHz. This results in a peak rate of 1600 million operations per second.

4.1 Time-domain Beamforming

In [2], we found that, a multiple board FPGA-based CCM can outperform a multiple DSP system composed of SHARC DSPs for our example time-domain beamforming application. In this previous paper we described how each FPGA could essentially process at the rate of 6 SHARC DSPs, due to memory addressing issues and the fact that each FPGA can hold two PEs.

Since then we have evaluated the TI ’C6201 for the time-domain calculation. The TI DSP can complete a time-delay kernel operation every two cycles — limited by the three memory accesses and the two-level memory look-up required. At 200 MHz, its throughput is 100 million MACs/second with single-cycle access to external memory. With this assumption, it performs on par with a single 50-MHz FPGA. However, this only accounts for raw processing speed and doesn’t consider communications or memory interfacing overhead which the TI DSP might encounter. In the end, these issues may dictate a much slower solution since high-performance memory access such as would be needed requires sequential access patterns, something this algorithm does not exhibit.

Assuming that the FPGA solution was not limited to four memory ports (use a XC4085XL FPGA with 448 user-definable I/O pins instead of a XC4028XL), the FPGA could hold as many as five PEs. A single 50-MHz FPGA could then perform 250 million time-delay MACs per second with only a 30% logic utilization, giving the FPGA an 18X speed advantage over a 40 MHz SHARC and at least a two and a half times speed advantage over the ’C6201.

[6]At the time of this writing, our results were based on preliminary specifications for a part scheduled to begin production in early 1998. We place much less confidence (vs. that for the SHARC) in our comparison to this part due our limited understanding of its future functionality. However, we have included it as being representative of an important future VLIW-like DSP architecture.

4.2 Frequency-domain Beamforming

The core function in frequency-domain beamforming is a complex phase shift and accumulate (CPAC) operation. In the case of the DSPs, these operations can be implemented many ways: a complex multiply-accumulate (CMAC), a mixed polar/rectangular approach like the FPGA solution above, etc.

The SHARC can sustain complex MACs at a rate of four cycles per MAC — the complex MAC requires four multiplies and four adds. At 40 MHz, the SHARC ADSP21060 can do 10 million CMACs/second. With a total required phase-shift rate of 8 billion CPACs/second, 800 SHARCs would be required, each forming 12 beams.

The problem with this approach is the memory requirements for the SHARC, considering that the SHARC only has a 512 KB memory and it must store over 1.2 million complex weights (4.8 MB total). Thus, each SHARC must have fast access to *at least* this much external RAM. Normally, for this to work efficiently, each SHARC and its external SRAM must be decoupled from the shared multiprocessor bus, making interprocessor communication somewhat more inefficient.

Another option is to perform the calculation in mixed polar/rectangular form, much like the FPGA implementation. This will require the DSP to do the following: compute the phase shifts on the fly given Δt values from memory, perform the phase shift via addition, convert the polar result to a rectangular form using $cos()$ table look-ups, and, lastly, perform the accumulation. Using this approach, the SHARC would complete a CPAC about every eight cycles, leading to a rate of 5 million CPACs/second and requiring 1600 SHARCs but no external memories.

In contrast, the TI DSP can complete a CMAC every two cycles since it has two multipliers and up to six adders. With a complex MAC rate of 100 million CMACs/second (and ignoring memory access and communication overhead), the problem would require 80 ’C6201 DSPs but with 52 MB of external SRAM per DSP to hold all of the weights. Alternately, using the phase-magnitude approach, a ’C6201 requires five cycles per phase shift for a rate of 40 million CPACs/second. Thus, that approach would require 200 ’C6201 DSPs, each with about 256 KB of external SRAM.

In contrast to the DSP realizations of the complex phase shift and accumulate operation, the FPGA-based version can complete one CPAC per cycle due to pipelining. With a 40 MHz clock rate and two PEs per FPGA, each FPGA has a CPAC rate of 80 million CPACs/second. Thus, this example beamforming application requires only 100 FPGAs.

Table 1 summarizes and compares the chip and memory sizes of the implementations. Again, we believe these results are very optimistic since they neglect the communications and external memory access overhead of the DSP solutions and thus are overly generous to them.

Knowing that the actual price of the system is only partially influenced by part cost, we want to point out the cost of the different devices we compared. All prices for parts are recent prices quoted for small quantities of devices (99 or fewer). The price of the SHARC in small quantities is approximately $300. The price of a previous version of the TI DSP in similar quantities was about $200. Finally, the Xilinx XC4062XL costs about $1300. Thus, a $1300 FPGA replaces a minimum of $2400 of SHARC DSPs. The TI DSP looks attractive from a price performance viewpoint but it is still premature to do system level cost comparisons.

4.3 Exploiting Parallelism

Considering that the FPGAs execute at moderate clock rates of 40–50 MHz, the main reason the FPGAs can outperform or compare with the SHARCs and the ’C6201 for these applications is parallelism. The parallelism which our FPGA-based beamformers enjoy exist at two levels: fine- and course-grain parallelism.

4.3.1 Fine-grain Parallelism

Fine-grain parallelism is the parallelism due to concurrent execution of several operations within a processing element. For both time- and frequency-domain beamforming, the DSPs require several cycles to perform the kernel operations but the calculations are simple enough to fit on an FPGA in a pipelined form. The following factors contributed to these differences in fine-grain parallelism:

- custom address-generation,
- customizable computation units,
- memory depth.

For the delay-sum beamforming case, the data access patterns are not easily calculated using regular intervals (or strides) or are otherwise outside the immediate capabilities of the DSPs’ address generators. The FPGA implementations, of course, can have custom address generators which can potentially provide a new address every cycle. Custom-address generation logic was one main reason the FPGA-based PEs are more efficient at delay-sum beamforming.

For frequency-domain beamforming, the main reason for the multi-cycle nature of the DSPs’ implementations of the kernel CMAC and CPAC is the lack of enough or the proper kind of functional units. For instance, if the DSPs could perform four multiplies, four adds, and two (or four, for the SHARC) loads per cycle, the processors might be able to perform the CMAC in a single cycle. For the FPGA-based PEs, the proper collection of functional units to perform the complex phase shift can all be placed in the processor to allow for single-cycle, pipelined operation.

Also, the ability of the FPGA to simultaneously generate the phase-shift values as needed in parallel with the other operations provides an additional advantage over the DSP systems—a dramatic decrease in memory requirements. Again, since the designer does not have the flexibility of changing the available functional units for the DSPs, the DSPs cannot perform the same function without suffering increased execution overhead.

A related issue in fine-grained parallelism is the amount of memory which the FPGAs and DSPs can interface with. For the frequency-domain application using CMACs, the DSPs cannot hold all of the data internally in their on-chip RAMs, the memories with which their highest computation rates can be achieved. For the SHARC DSP, this is clearly a shortcoming, since an external memory often resides on the shared multiprocessor bus, leading to memory and bus contention and, thus, more cycles per kernel operation. This is less of a problem for the TI DSP since it can interface directly with external RAM, but as we have mentioned, the amount of data required and interfacing issues lead to inefficiencies in the DSPs memory system.

Since DSPs generally have a limited amount of fast on-chip RAM, this is a common situation, meaning that for many applications the DSPs must rely on external memory and/or process data in blocks, leading to both memory and communication overhead. The amount and depth of the memory per FPGA is more of a system- or board-level design decision, but with a properly designed board, each FPGA can have appropriately deep memories for the target applications. Thus, with the flexibility of designing with FPGAs, memory depth does not have to be an issue.

4.3.2 Course-grain Parallelism

In contrast to the other form of parallelism, course-grained parallelism reflects the number PEs which can be implemented per FPGA. The main issues which affected this class of parallelism are:

- calculation simplicity,
- customizable I/O,
- CORDIC units.

Device	Style	Chip Count	External Memory Per PE (MB)	Chips vs. FPGAs	Total System Memory vs. FPGA Version
SHARC	rectangular	800	4.8	8x	38x
SHARC	polar	1600	0	16x	0x
’C6201	rectangular	80	52	0.8x	42x
’C6201	polar	200	0.25	2x	0.5x
FPGA	polar	100	0.50	-	-

Table 1: Comparison of FPGA and DSP Frequency-domain Beamformers

For both domains, the computations were simple, requiring relatively little area using today’s FPGAs. Consequently, several PEs can fit on a single FPGA.

Reflecting on memory bandwidth, the DSPs are generally limited to two memory accesses per cycle. For a given board design, an FPGA might be able to support as many as 10 external memory accesses per cycle to 256Kx16 SRAMs. This is a very key issue for our beamforming applications. If only one or two external memories were available per FPGA, the only parallelism which the FPGA designs could exploit would be fine-grained parallelism. Quite simply, the more memory ports accessing memories of adequate depth, the more PEs can be potentially placed on an FPGA. Though this may seem quite obvious, many CCM platforms do not provide more than one external memory port per FPGA, leading to poor suitability for beamforming applications.

Further, the availability of small on-FPGA memories is an advantage DSPs do not enjoy. Both Xilinx CLB RAM and Altera EABs have been useful in our studies. In the case of the frequency domain beamformer on Xilinx, the frequencies used in calculating the phase shifts “on-demand” can be stored on-chip, thus not requiring another port to external memory (and an associated fetch operation).

For frequency-domain processing, the CORDIC units provide significant area savings. As mentioned above, CORDIC is very amenable to high-performance FPGA implementations. Since pipelined array multipliers dominate the size of the PE, replacing four multipliers and two additions with one CORDIC and an addition reduces the area of a single PE by about 75%. This provides a way of overcoming some of the time-area disadvantages FPGA-based multipliers experience when compared to those of VLIW and other processors [8].

5 Conclusions and Future Work

Before embarking on this work with FPGAs and sonar processing, we believed that, for applications composed of MAC operations, complex arithmetic, and other common signal processing operations, DSPs would outperform FPGAs. This work demonstrates FPGAs are effective in signal processing applications, especially, when the applications meet the criteria mentioned in [1], when DSPs require multiple cycles for the kernel operations, when DSP on-chip memories are too small, and when the simplicity of the calculation allows the FPGA to fit several processing elements per FPGA. With an FPGA’s ability to support several processing elements per chip and considering the relatively moderate clock rates of most DSPs, there should be many cases where FPGAs are a better choice than integer DSPs. Also, this work demonstrates the great importance of CORDIC for FPGA-based signal processing.

Our ongoing work is directed at completing a system-level analysis on the differences between a complete design based on the architecture shown in Figure 1 and 2 and one based on DSPs. This will include communications, power, host-based control and interface issues.

Acknowledgments

This effort was sponsored the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government.

We would like to add additional thanks to Robert Bernecky of the Naval Underwater Warfare Center (NUWC) for his help in understanding beamforming and its related challenges as well as for providing parameters for realistic beamforming applications.

Additionally, we would like to thank Michael Rytting for the work he performed in providing performance and sizing statistics for the time- and frequency-domain beamformers.

References

[1] W. Mangione-Smith and B. Hutchings, “Configurable computing: The road ahead,” in *Reconfigurable Architectures: High Performance by Configware* (R. Hartenstein and V. Prasanna, eds.), Microsystems Engineering Series, (Chicago), pp. 81–96, IT Press, 1997. Proceedings of the Reconfigurable Architectures Workshop (RAW ’97).

[2] P. Graham and B. Nelson, “Fpga-based sonar processing,” in *Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’98)*, ACM/SIGDA, ACM, 1998. This paper has been accepted for presentation at FPGA ’98.

[3] C. Seitz, “A prototype two-level multicomputer.” http://www.myri.com/research/darpa/96summary.html, 1996. Project Information on DARPA/ITO project.

[4] T. Boggess and F. Shirley, “High-performance scalable computing for real-time applications,” in *Proceedings of the 1997 6th International Conference on Computer Communications and Networks (ICCN)*, (Piscataway, NJ), pp. 332–335, IEEE, IEEE, September 1997.

[5] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su, “Myrinet—a gigabit-per-second local area network,” *IEEE Micro*, vol. 15, pp. 29–36, February 1995.

[6] B. V. Veen and K. Buckley, “Beamforming: A versatile approach to spatial filtering,” *IEEE ASSP Magazine*, vol. 5, pp. 4–24, April 1988.

[7] D. H. Johnson and D. E. Dudgeon, *Array Signal Processing: Concepts and Techniques*. Prentice Hall Signal Processing Series, Englewood Cliffs, NJ: Prentice-Hall, 1993.

[8] O. T. Albaharna, P. Cheung, and T. J. Clark, “On the viability of FPGA-based integrated coprocessors,” in *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (J. Arnold and K. L. Pocek, eds.), (Napa, CA), pp. 206–215, Apr. 1996.

[9] J. Volder, “The cordic trignometric computing technique,” *IRE Transactions on Electronic Computers*, vol. EC-8, pp. 330–334, September 1959.

[10] J. S. Walther, “A unified algorithm for elementary functions,” in *Proceedings AFIPS Spring Joint Computer Conference*, pp. 379–385, AFIPS, 1971.

[11] Y. H. Hu, “CORDIC-based VLSI architectures for digital signal processing,” *IEEE Signal Processing Magazine*, pp. 16–35, July 1992.

[12] R. Andraka, “A survey of cordic algorithms for fpga based computers,” in *Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’98)*, ACM/SIGDA, ACM, 1998. This paper has been accepted for presentation at FPGA ’98.

[13] J. Ianniello, “A matlab version of the kraken normal mode code,” Technical Memorandum 94-1096, Naval Undersea Warfare Center, New London, CT, 1994.

GIGAOP DSP ON FPGA

Brad L. Hutchings and Brent E. Nelson

Brigham Young University
Dept. of Electrical and Computer Eng.
459 CB
Provo, UT 84602
hutch@ee.byu.edu, nelson@ee.byu.edu

ABSTRACT

DSP algorithms such as sonar beamforming and automated target recognition, are a good match for FPGA technology due to their regular structure, available parallelism, pipelineability, and modest data word sizes. FPGA implementations of these applications outperformed their DSP and microprocessor counterparts by factors ranging from 10X on up with an equivalent sustained computational rate of more than 2 GOps/second per FPGA. This paper first describes each application and derives its computational requirements. The mapping process for each is then described followed by an analysis of the relative contributions to performance from pipelining, data parallelism, and memory usage.

1. INTRODUCTION

FPGA technology provides attractive solutions for a range of applications areas. With their inherent reprogrammability, FPGA's exhibit characteristics normally associated with programmable processors. At the same time, they often provide solutions with order-of-magnitude performance advantages over programmable processors. This combination of flexibility and performance puts them in a unique place between processors and ASIC's.

Like all semiconductor products, FPGA's have historically followed Moore's Law where the number of components on a chip is doubling every 18 months. This is shown in Figure 1 for Xilinx devices. Similar density increases have been demonstrated by other vendors as well. The rightmost data point is the recently-announced *Virtex-II*, containing the equivalent of more than $10,000,000$ gates.

Until a few years ago it was difficult to contemplate the use of FPGA's for many DSP computations due to the need for wide-word arithmetic and floating point computations. However, as FPGA's continue to grow in density this is now feasible. Put another way — while FPGA's are not *catching up* with ASIC's in terms of raw performance, they have crossed the density threshold required for use in many advanced DSP applications. As such, factors like time-to-market

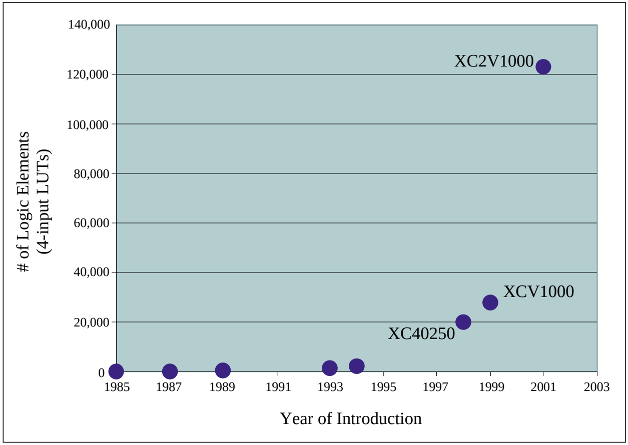


Fig. 1. FPGA Density

and NRE costs are combining to make FPGA's competitive in many cases with ASIC's for application-specific DSP solutions.

Algorithms typically implemented on DSPs generally have the following characteristics: very large amounts of exploitable parallelism, modest data word sizes (16-32 bits) and relatively simple control algorithms that can often be statically scheduled. Such algorithms are also well suited to modern FPGA devices. The sheer size of modern FPGA devices makes it feasible to exploit much of that available parallelism. Small, fixed data word sizes make it feasible to implement high-performance data paths that can be customized to specific phases of computation. The simple control schemes used in these algorithms can be directly implemented as fast, customized state machines of moderate complexity. Finally, control and data-path circuitry can be implemented with distinct circuitry specifically developed for these separate purposes. This allows the data-path to operate at 100% efficiency with no interference from control. This is in contrast to DSPs where control instructions interfere with the data-path operation. For example, in the applications described in this paper, custom data-paths which consist of deeply pipelined chains of operations are constructed in FPGA's. The result is that relatively complex inner loop computations can be computed at a throughput of one per cycle without interference from concurrently executing control circuitry.

To demonstrate these principles the body of this paper describes two very different DSP applications implemented in FPGA's. The first is an image processing application where a thresholded binary image is manipulated using morphological operations such as dilation and erosion to identify regions of interest in an automated target recognition system. The second is a matched-field frequency-domain sonar beamformer. The first is dominated by bit-level operations and contains a minimum of control circuitry (it is implemented by a pipeline of morphological operators). The second is dominated by complex arithmetic and a nested-loop control structure. In spite of the differences between the two algorithms they combine to illustrate the principles from above and provide examples of the high computational rates achievable with FPGA technology.

2. BINARY MORPHOLOGY

Binary morphology consists of a set of operations used to find, enhance and/or remove certain geometric features in binary images [1]. In our case, binary morphology implements a Focus of Attention (FOA) algorithm that serves as the first data-filter stage in an automated target recognition (ATR) algorithm to find and pass on only those regions of the image most likely to contain a potential target. Binary morphology can be used this way because it can be used to detect image regions that contain shapes that are a certain size, or that have a certain aspect ratio, etc. Prefiltering the data this way improves performance by dramatically reducing the amount of image data that need to be processed by the computationally demanding target recognition algorithms.

The most important operations for our purposes are dilation, erosion, and the hit-and-miss transform; all these operations can be computed using a computational process akin to image convolution. The inputs to this process are all binary: the image to be transformed (the input image), and a small image kernel (referred to hereafter as the *structuring element*) that for our purposes is 3 x 3 pixels in size. The output of this process is a transformed image of approximately the same size as the input image. Pixels in the output image are computed by "placing" the structuring element at each pixel location in the input image and logically comparing all pixel values of the structuring element against the corresponding pixel values in the input image. Figure 2 depicts simple examples of dilation, erosion, and the hit-and-miss transform: on-pixels are black, off-pixels are white.

Implementing the FOA algorithm is done by "chaining together" many binary morphological operators, one after the other. When implemented in software, FOA is implemented as a succession of function calls, where each function call implements one morphological operation. In FPGA hardware, FOA is implemented by creating a deep image-processing pipeline which consists of many hardware modules (each performing a single morphological operation) chained together. To ease programming complexity, a generic hardware module has been developed that can implement dilation, erosion, or hit-and-miss operations. This module is shown in Figure 3. This hardware module consists of delay lines and registers that align the incoming serial image data stream into a spatial form where 3 x 3 neighborhoods can be operated on. Also shown in the figure are the Template Matcher and Final Calc blocks that actually compute the value of the output pixel; these contain programmable ROM locations that determine which morphology operation is performed.

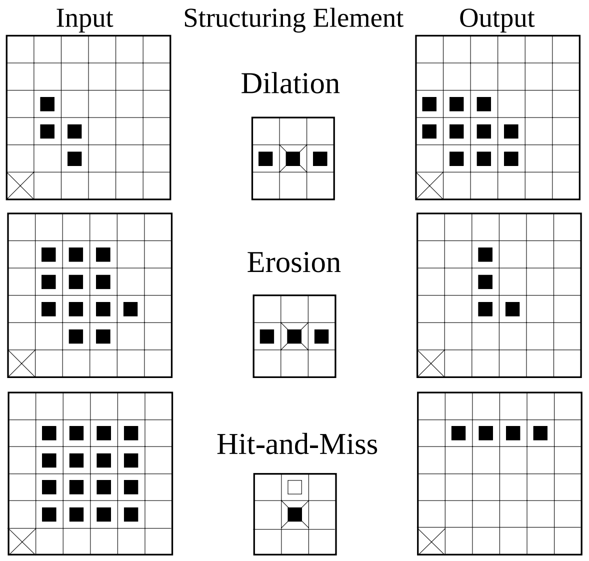


Fig. 2. Morphology Examples

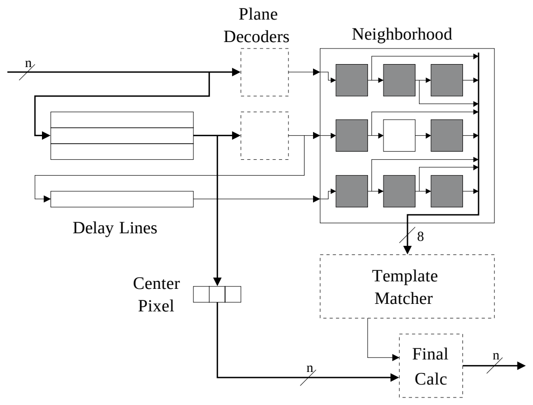


Fig. 3. Generic Morphology Operator Module

2.1. Performance Comparisons

In this section, FOA performance of the FPGA implementation described above will be compared against a highly optimized software implementation currently in use at Sandia

Device	Clock Rate	Clock Count	Time	BOPC
G4	400 MHz	108,000,000	.24s	22.1
XCV2000	50 MHz	1,181,053	.023s	2219.8

Table 1. FPGA and Software Performance Comparison

National Labs. This comparison will use bit-ops per clock cycle (BOPC) as the figure of merit. For this comparison, a bit-op is defined to be a single binary, boolean operation (AND, OR, etc). In this analysis, the effective BOPC rate is computed by: 1) carefully examining the source software and counting the total number of bit-ops, 2) counting the total number of clocks required by each implementation (FPGA and software) by running the applications and measuring runtimes, and 3) dividing the total bit-op count by the total number of clocks. The final value represents the number of effective bit-ops that are computed per clock cycle. A typical FOA application requires 2275 bit-ops per pixel with the total bit-ops for a (1024 x 1024) image being 2,385,510,400. Note that this bit-op count only takes into consideration bit-ops that contribute directly to the computation of morphological operations. Overhead due to address computation, branching, etc., is not considered in this bit-op count.

Table 1 compares the performance of the software and FPGA implementations. The FPGA implementation achieves a very high BOPC count because it is organized as a very deep pipeline of concurrently operating hardware modules (see Figure 3). This organization minimizes control overhead by using line delays to address pixels to allow each hardware module in the pipeline to perform a morphological operation on a single pixel every clock cycle. This is in contrast to the G4 which must share computational resources for both control (address calculation, branching, etc.) and computation of the morphological operations. Although the FPGA implementation achieves a 100x higher BOPC count than the G4, it achieves a clock rate that is about 1/10 that of the G4, resulting in an overall throughput that is approximately 10x that of the G4. In the end, this application achieves high performance because it uses static scheduling (which is a result of connection of line delays) and many customized functional units (the generic hardware modules) which when combined allow the hardware to exploit data-level parallelism to compute at a very high rate.

3. PASSIVE BEAMFORMING

Beamforming is used to determine the direction-of-arrival (DOA) of a signal and has use in RADAR, SONAR, and acoustic applications. It takes advantage of the fact that a signal arriving at an array of sensors will arrive at each sensor with a different phase. Knowledge of the array geometry makes it possible to test for a signal arriving from a particular direction by appropriately delaying each sensor's response (to bring each received copy of the signal into phase with all the others) and summing. Since the signals of interest are periodic, maximum power will result when the delayed sensor responses are in phase.

the

the sensor number,

Frequency domain techniques are commonly used to beamform selected frequencies of received signals. To do this, an FFT of the sensor data is first computed and the following algorithm executed:

```
for d = 0 to numDirections
  for f = 0 to numFrequencies {
    for (k=0;k<numSensors;k++)
      sum[d, f] += fftData[d, f] *
                   steeringWeights[d, f];
  }
```

where the *fftData* and *steeringWeights* are complex values. A full treatment of beamforming techniques can be found in [2].

A problem with the above computation is the storage of the steering weights. Consider a typical problem with $2,500$ directions, 256 frequencies, and 400 sensors. The storage required for steering weights in this case would be $1GB$. A key observation is that the frequency domain computation outlined above is essentially equivalent to a time-delay computation — the signals of interest are delayed prior to summing. However, in the frequency domain approach, the steering weight accomplishes this by phase rotating the FFT data.

The approach taken in our design is to store time delays for each direction and sensor. Steering weights (phase adjustment terms) are formed on-the-fly via a multiplication of the time delay with the frequency of interest. This phase adjustment is then added to the phase term of the FFT data (the FFT data has been pre-converted to polar form), the rotated data is then converted to rectangular form and summed. This is shown in the following:

```
for d = 0 to numDirections
  for f = 0 to numFrequencies {
    for (k=0;k<numSensors;k++) {
      phaseAdjust = delay(d,k) * f;
      phase = fftPhase[k, f] + phaseAdjust;
      mag = fftMag[k, f];
      sum[d, f] +=
        polarToRectangular(mag, phase);
  }
```

This reduces the storage required for steering weights from $1GB$ to $4MB$. The complex multiply required in the original computation is replaced by a scalar multiply, a scalar addition, and the *polarToRectangular()* function (implemented as a hardware CORDIC rotation). The result is approximately a $2X$ area reduction. The data-path for this computation is shown in Figure 4.

The above computational kernels have been employed in the construction of a number of beamformers including

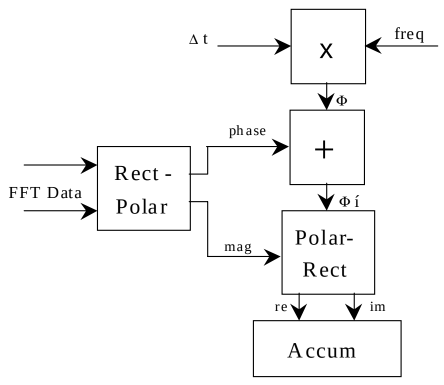


Fig. 4. Frequency-Domain Beamformer data-path

those for both SONAR and air-acoustic environments. The latest is a two-stage matched-field SONAR beamformer designed for shallow water environments. The first stage is a k-ω beamformer which beamforms multiple rays (direct path and those bouncing off the bottom and surface). The first stage localizes the target to an ocean voxel and the second stage then does a sub-voxel interpolation beamformer computation to determine the precise 3-D location of the target. Both stages employ algorithms with inner loop computations similar to the design shown in Figure 4.

The computation was mapped to a SLAAC1b PCI board consisting of a Xilinx 4085 (PE0), two Xilinx 40150 FPGA's (PE1 and PE2), and 10 SRAMs. The system runs at a 50 MHz clock rate. The width of the majority of the arithmetic operations performed is between 12 and 16 bits. PE0 interfaces with the host and does the k-ω beamforming once a second. It sends its results to PE1 and PE2 which perform the subvoxel beamforming. A total of eight subvoxel beamformers fit into the combination of PE1 and PE2 and operate in parallel. Thus, PE1 and PE2 perform 400M inner loops per second. The inner loop calculation represents about 10 operations giving a total delivered computation of about 4 GOp/second. The FPGA implementation was compared to that running on a variety of machines including Pentium-II and Pentium-III machines, HP PA-RISC workstations, and G4 Power PC's. The fastest performing machine was a 552 MHz PA-RISC workstation and its runtime was 18 times as long as the FPGA. The slowest machine was a 400 MHz Pentium-II machine with a runtime 83 times as long as the FPGA.

3.1. Beamformer Analysis

The above beamformer design is typical of our experience — an order-of-magnitude board count advantage of FPGAs over processors has been typical when considering embedded systems packaging options. This is mainly due to two factors. First, the amount of parallelism available in beamforming is very, very high. Our design places four processing pipelines on each Xilinx 40150 part. However, the design has adequate parallelism to directly scale *without modification* to FPGA parts which would hold two hundred processing pipelines each — a $50\times$ increase. Second, the control structures required in beamforming implement simple statically-scheduled nested-loop computations. The lack of data-dependent control makes pipelining of the entire design possible down to the LUT level resulting in a high system clock rate.

4. CONCLUSIONS

Amongst all the important algorithm characteristics listed in the introduction, the two most important by far are unbounded parallelism and pipelineability (the lack of cyclic data dependencies). Unbounded parallelism is important because FPGAs typically achieve a clock rate about 1/10th that of a microprocessor implemented in the same technology; this means that an FPGA implementation must exploit about 10x more parallelism just to break even with a high-performance microprocessor. Achieving a 10x *throughput* increase over a microprocessor requires that the FPGA implementation exploit about 100x more parallelism as shown in the applications discussed above. Lack of cyclic data dependencies is also essential because the relatively slow, programmable interconnect used in FPGAs demands the use of pipelining to achieve high clock rates. In addition, such pipelining provides an effective way to exploit much of the parallelism available in applications.

These results demonstrate the feasibility of Giga-Op DSP on FPGAs. Design effort was not herculean and was similar to writing high performance embedded software. High performance was possible because both applications exhibited unbounded parallelism and a lack of cyclic dependencies. Moreover, the relatively simple control schemes used in these applications could be implemented with dedicated, statically-scheduled circuitry (fast, simple finite state machines) that could operate at the same rate as the highly customized data-path circuitry enabling 100% utilization of the data-path.

Finally, note that many important applications in image and signal processing exhibit both unbounded parallelism and few or no cyclic dependencies (either in their entirety or for some important kernels) making them feasible candidates for FPGA implementation. Because of this, we can expect to see the use of FPGAs in DSP applications to grow significantly.

5. REFERENCES

[1] Milan Sonka, Vaclav Hlavac, and Roger Boyle, *Image Processing, Analysis, and Machine Vision,* PWS Publishing, 1999.

[2] N. L. Owlsey, *Array Signal Processing,* Prentice-Hall, 1985.

High Performance Applications on Reconfigurable Clusters

by

Zahi Samir Nakad

Thesis Submitted to the Faculty of
Virginia Polytechnic Institute and State University
In partial fulfillment of the requirements of the degree of

Masters of Science

In

Computer Engineering

November 13, 2000
Blacksburg, Virginia

Keywords: Reconfigurable Computing, FIR Filters, Constant Coefficient Multipliers, FPGA

High Performance Applications on Reconfigurable Clusters

Zahi Samir Nakad

Committee Chairman: Dr. Mark Jones
The Bradley Department of Electrical Engineering

Abstract

Many problems faced in the engineering world are computationally intensive. Filtering using FIR (Finite Impulse Response) filters is an example to that. This thesis discusses the implementation of a fast, reconfigurable, and scalable FIR (Finite Impulse Response) digital filter. Constant coefficient multipliers and a Fast FIFO implementation are also discussed in connection with the FIR filter. This filter is used in two of its structures: the direct-form and the lattice structure. The thesis describes several configurations that can be created with the different components available and reports the testing results of these configurations.

To (Samir, Layla, Youssef, Wassim) Nakad

Acknowledgements

This thesis work would have never been accomplished without the valuable guidance and support of Dr. Mark Jones. I am extremely thankful for his patience and guidance throughout my work in the CCM lab, in his classes, and especially in the last phase of editing, reediting, and re-reediting.

I am extremely thankful to Dr. Peter Athanas for his help and guidance. My gratitude goes to him for his technical support and suggestions that made my work a success.

My thanks go out to Dr. James Armstrong for serving on my defense committee and in taking the time to read my thesis.

I would also like to thank all the people in the CCM lab who have taken from their time to offer pointers and support throughout my work. My thanks go out to: Louis Pochet, Luke Scharf , Dennis Collins, Santiago Leon, Ryan Fong, and all the friends I have in the lab.

I am also grateful to Claudia Rente who had the daunting task of editing my grammar in this work.

Last and not least, I want to extremely thank my parents for their invaluable support throughout my academic life that is still going on. Their moral support was immeasurable but living away from them has proved to be extremely hard.

Table Of Contents

List Of Figures

List Of Tables

Chapter 1: Introduction

Many problems faced in the engineering world are computationally intensive. These problems can be solved on general-purpose computers, Application Specific Integrated Circuits (ASICs), or Configurable Computing Machines (CCMs). In order to allow flexibility, a significant amount of the capabilities of the hardware in general-purpose machines is lost, preventing the use of certain implementation optimizations. ASICs can implement all optimizations needed, but implementation is the final and irreversible step; no changes to the design can be made after implementation. CCMs offer a compromise, where all required optimizations can be implemented and the flexibility of change or reconfiguration is possible [1], [2].

1.1 Thesis Contributions

The main contributions of this thesis are based on developing a fast and reconfigurable FIR filter on a commercial configurable computing accelerator. A lack of I/O speed in data entry and acquisition from the WILDFORCE board (discussed in Chapter 2) hinders running designs at higher speeds. A Fast FIFO was created that took advantage of the fast DMA accesses to the memory of the Processing Elements (PE) on the board. By providing new data paths this Fast FIFO allows implemented designs to run at higher speeds. The other difficulty in developing fast FIR filters is the area cost of the multipliers; constant multipliers were introduced in this phase. Because optimization of one of the coefficients helps in optimizing the multipliers, they can offer higher speeds and smaller area consumption.

This thesis compares and contrasts different hardware implementations of a FIR filter. The thesis studies an implementation that allows coefficients of the filter to be changed through the data path, using JHDL multipliers. This contrasts with an implementation that uses constant multipliers but requires full reconfiguration of the filter. These two implementations will also be provided in two structures of the FIR filter: the direct form and the lattice structures. Another comparison point is passing data through the filter using either the regular FIFOs or the Fast FIFO.

1.2 FIR Filters

Filtering using FIR (Finite Impulse Response) digital filters is a computationally intensive problem. If general-purpose computers are used to solve this problem, many of the hardware's capabilities will not be used. However, designing a new ASIC for each response needed is expensive and time consuming. Having a filter implementation where the filter response can be reconfigured on the fly is very useful. FIR filters can be used in many applications such as wireless in-door channel modeling [2]. Another example is the use of the lattice structure of the FIR filter in digital speech processing and in implementing adaptive filters [3].

This thesis studies the implementation of FIR filters on CCMs. The FIR filters discussed are implemented in the direct and lattice form. Two methods of reconfiguring the response of the filters are studied. The first is runtime, using information sent through the data path. The second makes use of constant coefficient multipliers; the response is changed with full synthesis of the design.

1.3 FPGA Boards

FPGAs (Field Programmable Gate Arrays) placed on boards that interface them to PCs, is one type of architecture where the above-mentioned characteristics of CCMs are found. The WILDFORCE and SLAAC1 boards are two such boards. The WILDFORCE board contains five FPGAs, referred to as PEs (Processing Elements). A systolic bus connects four of these PEs (PE1 – PE4). Using the systolic bus, these four PEs can be used in a pipeline-like structure. The SLAAC1 board contains three PEs, which can also be used in a pipeline-like structure. Communication with the outside world, on both boards, is through FIFOs or through PE memory access [4], [5].

The hardware WILDFORCE FIFO speed in communicating data is slow, especially when compared with the DMA mode of accessing memory. This thesis studies a method called the Fast FIFO which makes use of the DMA mode and provides an interface close to that of a FIFO. The Fast FIFO provides a significant increase in the speed of accessing data, at the price of using computational resources (PE0 and PE4) [4].

1.4 Multipliers and Area Consumption

Limited space on FPGAs is a problem that should be addressed in most designs to be implemented. Multipliers consume a large amount of space. Multiplying the input data with coefficients in each tap is required in the FIR filter implementation. These coefficients do not change unless the response of the filter is to be modified. The infrequent change in the constants makes constant multipliers attractive for consideration. Much space can be saved by the use of constant multipliers, because knowing the coefficients introduces many optimizations to the multiplier. The use of Booth's algorithm and inverting the constant are two optimization techniques examined[6]. The reported results manifest a significant increase in speed and decrease in area consumption.

1.5 Thesis Organization

Chapter 2 of this thesis provides the background material on which the following work is based. Chapter 3 introduces the constant coefficient multipliers that were created and then used in the implementation of the FIR filter. Chapter 4 explains the direct form and lattice implementations of the FIR filter and it also discusses data-path and re-synthesis reconfiguration of the filter. Chapter 5 describes the Fast FIFO implementation and its uses. Chapter 6 describes how all the pieces integrate and reports the results of running the filter under various implementations. Chapter 7 concludes the work and provides future insight on further developments.

Chapter 2: Background

Speech processing, adaptive filtering, and channel modeling are examples of applications where FIR filters are used [2], [3]. Filtering using a FIR filter is a computationally intensive task and typically requires floating point multiplication and addition.

The following section provides the background used in the work to implement a fast reconfigurable FIR filter. Background material on the theory of the FIR filter, the constant coefficient multipliers, the WILDFORCE and the SLAAC-1 boards, and the ACS_API is provided.

2.1 FIR Filters

"Finite impulse response (FIR) filters have been referred to in the literature as moving average filter, transversal filters, and nonrecursive filters. [7]" The main attractive characteristics of this kind of filter are that it is stable and can be made to have an exact linear phase [7], [8]. Coefficient quantization does not affect the response of a filter with linear phase as severely as in other cases; the effect is only in magnitude as opposed to having the effect of the quantization in phase also [3].

Direct-form, cascade-form, frequency-sampling and lattice structures are different forms of implementing the FIR filter in hardware, and are discussed in [3]. Direct-form and lattice are the two structures that are implemented in this thesis. The Direct-form was chosen for its simplicity. This form follows directly from the nonrecursive difference equation (discussed below). This simplicity renders debugging easier because all the intermediate values are easily followed, and the hardware needed to implement the filter is straightforward. The lattice structure was chosen for its cascading capability; this structure can be viewed as basic units that can be "strung" together without any regard to the number of stages or the position of a certain unit in the whole structure.

As its name implies, the Direct-Form structure follows exactly from the nonrecursive difference equation:

$$y(n)=\sum_{k=0}^{M-1} h(k)x(n-k) \tag{1}$$

The structure is shown in Figure 1, which shows that there are *M* multiplications and *M-1* additions in this implementation. M is the number of stages; we see that the output of the filter is based on the weighted sum of the last M-1 values in addition to the current value (hence the M multiplications) [3].

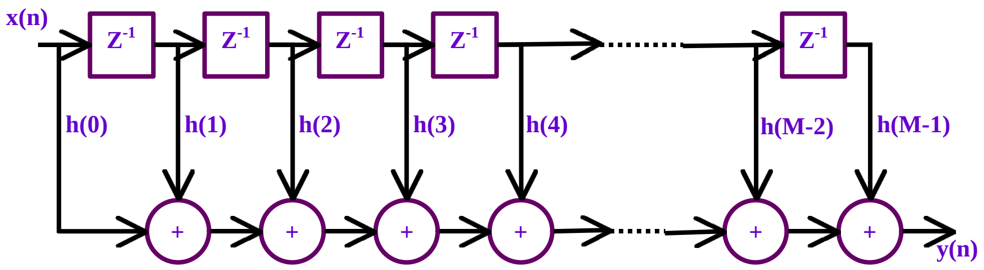


Figure 1 Direct-Form structure of an FIR filter (adapted from Figure7.1 [3])

The lattice filter is also derived from the nonrecursive difference formula, after some manipulation. In this case the response of the mth term, $h_m(0) = 1$, $h_m(k) = \alpha_m(k)$, and k = 1,2,3,…,m. $\alpha_m(0)$ was taken to be 1 for mathematical simplicity. With these changes, the formula for the filter becomes:

$$y(n)=x(n)+\sum_{k=1}^{m} a_m(k)x(n-k) \tag{2}$$

The coefficient m, that denotes the number of stages in the filter is called the degree of the polynomial $A_m(z)$ representing the filter response; in this case

$$A_m(z) = 1 + \sum_{k=1}^{m} a_m(k) z^{-k} \tag{3}$$

If we consider an FIR filter where m = 1, then

$$y(n) = x(n) + a_1(n)x(n-1) \tag{4}$$

If we call the coefficients of the lattice structure K_m, then the lattice structure filter with the same response will have $K_1 = \alpha_1(1)$. The structure of such a lattice filter is shown in Figure 2 [3].

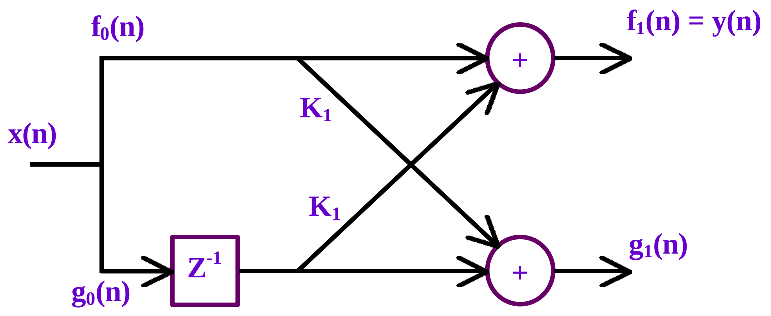


Figure 2 Lattice structure FIR filter with one degree (adapted from Figure7.9 [3])

The above figure shows that $f_1(n)$ corresponds to equation (4) of the FIR filter of degree 1. Considering an FIR filter with m = 2; the response is:

$$y(n) = x(n) + a_2(1)x(n-1) + a_2(2)x(n-2) \tag{5}$$

To get a similar result we cascade two lattice stages to get the structure shown in Figure 3 [3].

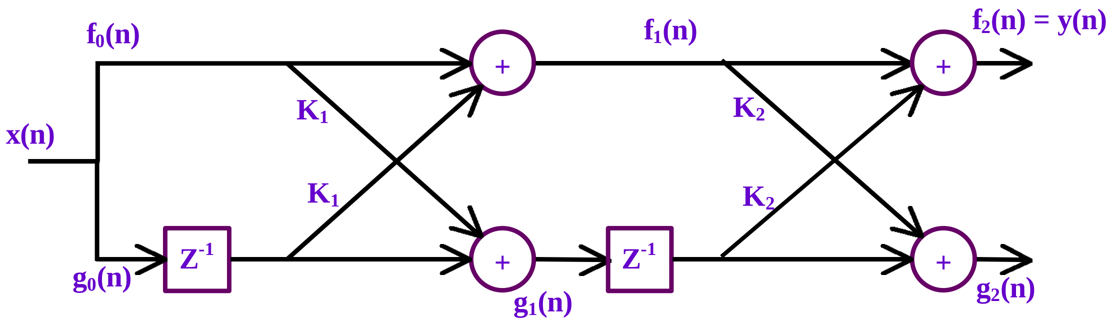


Figure 3 Lattice structure FIR filter with two stages (adapted from Figure 7.10 [3])

From the figure above we can deduce the following formulas:

$$f_1(n) = x(n) + K_1 x(n-1) \tag{6}$$

$$g_1(n) = x(n-1) + K_1 x(n) \tag{7}$$

$$y(n) = f_2(n) = f_1(n) + K_2 g_1(n-1) \tag{8}$$

$$g_2(n) = g_1(n-1) + K_2 f_1(n) \tag{9}$$

Putting equations (6) through (9) together, we end up with the following result:

$$y(n) = x(n) + K_1(1+K_2)x(n-1) + K_2 x(n-2) \tag{10}$$

This equation is the same as (5) with the following relationships between the α and K constants:

$$a_2(2) = K_2 \qquad a_2(1) = K_1(1+K_2) \tag{11}$$

The stages in the lattice structure can just be added until the required degree is established. Equations (8) and (9) can be generalized by substituting m for 2 and m-1 for 1 in the subscripts [3].

.

2.2 Constant Coefficient Multipliers

Implementing multipliers in hardware requires a significant amount of area; an example is the array multiplier (fixed point multiplier) used in the JHDL modgen. The area needed for the unpipelined version depends on the widths of the inputs, x and y, in the following formula: $area = y * (\lfloor x/2 \rfloor + 1)$ [9]. This space can be an obstacle in creating certain designs if they do not fit on the chip or silicon that is to be used. If one coefficient that is to be multiplied is known or is going to be fixed for a long time, designing constant multipliers can help in creating hardware that saves in space and time, due to the optimizations gained from knowing one of the inputs [6].

The more 1's present in the binary representation of the coefficient, the greater the number of additions that are needed to get the product. Booth's algorithm reduces the number of additions by replacing any group of consecutive 1's with a subtraction of the term corresponding to the least significant position of the 1's from the term corresponding to the bit position that is one higher than the most significant 1. Figure 4 displays the multiplication of 5 (0101) by 7 (0111), both with and without the use of Booth's algorithm.

```
Without Booth's algorithm:
        0101
        0111
        0101   Term1: corresponds to the 1 in bit position 0 in (0111)
       01010   Term2: corresponds to the 1 in bit position 1 in (0111)
      010100   Term3: corresponds to the 1 in bit position 2 in (0111)
      100011
With Booth's algorithm:
        0101
        0111
     1111011   Term1: 2's compliment of (0101) corresponding to position 0
     0101000   Term2: corresponds one bit higher than position 2
     0100011
```

Figure 4 Multiplication using Booth's Algorithm

In the multiplication in Figure 4, Booth's algorithm canceled one of the terms that was to be added. The number of ones in the sequence has no significance to Booth's algorithm; more savings are attained with this method if there are more 1's in consecutive positions [6].

Another technique can be derived from Booth's algorithm, based on observing the results of the following example: if we have a series of three consecutive 1's followed by another series of three consecutive 1's, and separated by a 0 (1110111), then using usual multiplication will result in six terms to add. Using Booth's algorithm on each of the series will result in four terms, two terms being created out of each series of consecutive 1's. Of these four terms, two are positive and two are negative. An important observation here is that the term constituting seven 1's (1111111) is really the term (1110111) as mentioned above, with (1000) subtracted from it. Thus, the multiplication with (1110111) can be substituted with a multiplication with (1111111 – 1000). The multiplication with the (1111111) can be solved by Booth's with two terms. Following the discussion above the whole multiplication can be resolved with the two terms from Booth's and then subtracting (1000). Figure 5 gives an example of multiplying 85 (1010101) with 119 (1110111).

```
Regular Multiplication           Booth's        New Technique
              1010101             1010101              1010101
              1110111             1110111              1110111
              1010101      11111110101011       11111110101011
             10101010          1010101000       10101010000000
            101010100      11101010110000       11110101011000
          10101010000      10101010000000       10011110000011
         101010100000      10011110000011
        1010101000000
       10011110000011

              6 Terms             4 Terms              3 Terms
```

Figure 5 Multiplication using Booth's Algorithm and the new technique

2.3 Floating Point Format

The floating-point representation used in this thesis is based on the IEEE 754 standard [9]. The 32-bit format, shown in Figure 6, has one bit for the sign (s), eight bits for the exponent (e), and 23 bits for the mantissa (m).

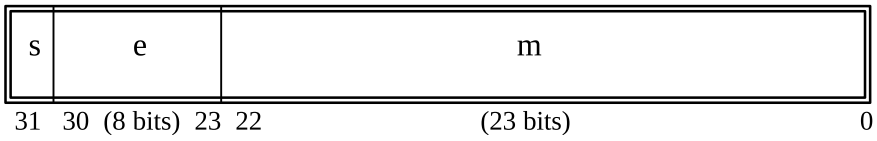


Figure 6 Floating Point Format (32 bits)

The value (v) of the floating-point is:

$$v = -1^{s} 2^{(e-127)} (1.m) \qquad (12)$$

There is a bias of 127 for the exponent in this 32-bit format. There is always a 1 at the most significant position of the mantissa. This 1 does not show in the representation; it is implicit. The floating-point number 0x3F800000 will be used as an example. Bit 31 or s is zero in this number so we know that it is positive ($-1^0 = 1$). The exponent e = 0111 1111 or 127, thus the $2^{(e-127)}$ term evaluates to $2^0 = 1$. The term m is all zeroes, so v = 1*1*(1.0) = 1 [1], [2], [10].

2.3.1 Floating Point Multiplication

Multiplication for the floating-point representation is done in separate steps. The first step is to isolate each of the three parts of the representation alone. The resulting sign is the “xoring” of the signs of the two numbers to be multiplied. The exponents are added. The mantissas are multiplied, and the decimal point position is then modified to normalize the result; the value of the exponent sum is fixed to represent the change needed in the normalization [1], [11].

2.3.2 Constant Coefficient Floating Point Multiplication

Floating-point multipliers require a large amount of space when implemented in hardware. The fixed-point multiplier used to multiply the two mantissas is the most expensive part of the floating-point multiplier. The same discussion used for constant coefficient multipliers applies here, if one of the terms is fixed. Thus, using a constant coefficient multiplier instead of a regular multiplier to multiply the mantissas can attain significant savings in the size of the floating-point multiplier.

2.3 WILDFORCE Board

The WILDFORCE is a product of Annapolis Micro Systems, Inc. This board holds five FPGAs (Field Programmable Gate Arrays) called Processing Elements (PE) and offers data communication between these five FPGAs and a controlling PC, usually referred to as the host [4].

2.3.1 The WILDFORCE Board Architecture

The WILDFORCE board constitutes five PEs, the first one is the control PE called CPE0. The other four are called PE1 up to PE4. Figure 7 shows how these PEs are placed on the board and the communication channels between them.

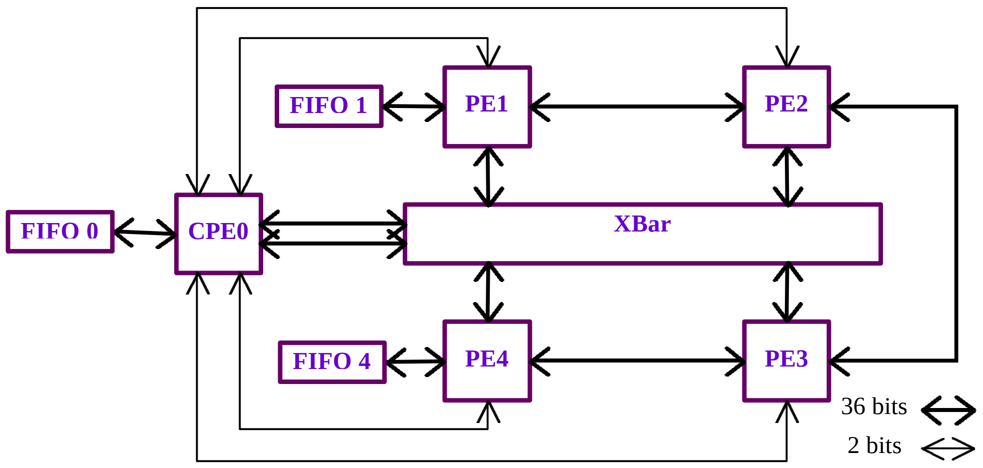


Figure 7 WILDFORCE Board Layout

For CPE0 to carry out its controlling tasks, it is provided with more connections to the communication channels on the board with respect to the other PEs. It has two interfaces with the crossbar, as opposed to one on the other PEs, and it is connected to each PE with a two-bit control bus. Figure 7 shows the connections that were mentioned with respect to CPE0; it also shows that PE1 to PE4 are connected to each other through a 36 bits wide bus [4], [9].

FIFOs are used to push data to the board and to pull them out. There are 3 FIFOs on the WILDFORCE board and they are connected to CPE0 (FIFO 0), PE1(FIFO 1), and PE4 (FIFO 4). FIFO 1 and FIFO 4 can be used as the end points of a computational path through the systolic bus and through four PEs. The FIFOs offer an easier way of pushing data to the board as compared with using memory. Using the FIFOs also saves on hardware needed to control addressing and capturing the data from the memory.

Each PE has its own separate memory that can be accessed by it and by the host program (discussed in the following subsection). DMA access is also available, this access provides very fast data transfer from the host program to the boards, as opposed to the FIFOs [4].

The crossbar offers connections between all PEs. It can be configured to any setting that is needed. A configuration file sets the configuration of the crossbar, the can also be changed during execution. From Figure 7, it is seen that using this crossbar, any PE can be connected directly to any other PE. The delay inside the crossbar is two clock cycles.

2.3.2 The Host Program

The host program runs on the local machine that controls the WILDFORCE board. This program makes use of the Application Programming Interface (API), which is provided with the board. Controlling the FIFOs, clock speed, memories, and PEs are examples of the host program’s capability through the use of the API [4].

2.4 SLAAC1 Board

The SLAAC1 board was developed by ISI-East [5]. There are several versions of this board, the difference is mainly in the type of the FPGA chips that are found on the board. The SLAAC1-A holds a XC4085XL and two XC40150XL Xilinx chips, while the SLAAC1-V holds three Virtex 1000 Xilinx Chips. The following sections will discuss the board irrespective of which version is being used [5].

2.4.1 SLAAC1 Board Architecture

The SLAAC1 board holds 3 PEs: the control PE (PE 0), PE 1, and PE 2. Figure 8 shows how the PEs are placed on the board and the communication channels between them.

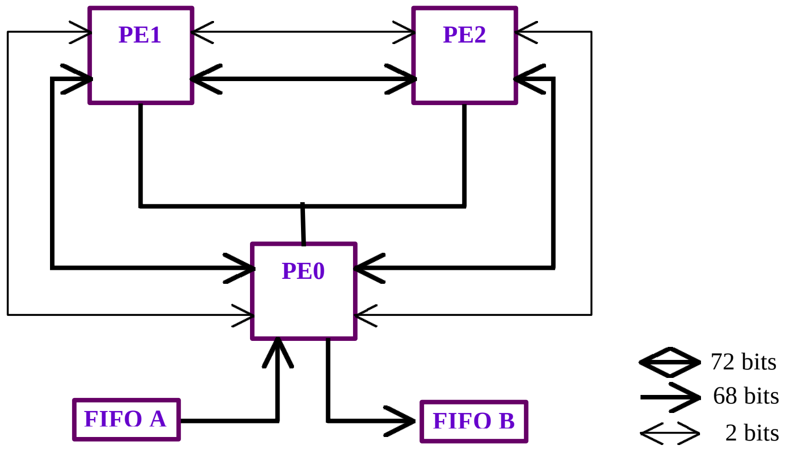


Figure 8 SLAAC1 Board Architecture

PE0 is the control Processing Element connected to the two other PEs through a two bit bus and a 72 bit data bus. This PE also takes care of data communication with the outside world through the FIFOs. FIFO A inputs data to the PE0 and through that to the other PEs, then PE0 takes care of writing to FIFO B, which pushes data to the host. PE0 can also communicate with the PEs through the crossbar. PE1 and PE2 are identical; they are connected to each other through the 72-bit bus, the two-bit bus, and the crossbar.

Each PE in the SLAAC1 has its own memory modules. The host program and the PE access these modules. The crossbar in the SLAAC1 has no configuration file, therefore the user specifies the connections to the crossbar and has to take care of contentions [5], [9].

2.4.2 The Host Program

The host program runs on the local machine, and controls the SLAAC1 board. Using the API provided with the board, the host program controls the FIFOs, clock speed, memories. Thus the API provides the host program and the user with all control needed over the SLAAC1 board [5].

2.5 ACS-API

The Tower of Power (TOP) in Virginia Tech is a good example of a parallel Adaptive Computing System (ACS). This system constitutes sixteen Pentium II machines, each equipped with the WILDFORCE Board. The sixteen machines are connected to each other through an Ethernet and a Myrinet [12] network. Developing designs on systems like the TOP requires more work than implementing on only one board. After creating the bit files that are to be downloaded on the PEs, the developer has to write network code to connect the boards together. This increase in work can prove to be tedious and time-consuming [13], [14].

The ACS-API offers a way out of creating the network application code. This API offers a high level environment where the user does not have to worry about the network communication code. The user only has to specify a channel between two nodes (a node refers to a board in the case of the TOP). The API also offers calls that will configure the boards, create the connections between them, as well as send and receive data between the boards without intervention by the user. This environment proves to be very helpful in developing designs that need more than one board because the developer will create the hardware needed on one board and then use the API to control as many boards as needed [13].

Another advantage of the ACS-API is that it is independent of the hardware it controls. This is a real help when using different types of boards together, the WILDFORCE and the SLAAC1, for example. The channel object in the environment can act as a buffer for data that is moving from one board to the other; this way the user does not have to worry about creating flow control mechanisms between the boards in the case of a fast sender and a slow receiver.

Chapter 3: Constant Coefficient Multipliers

The following chapter discusses the implementation of the constant coefficient multipliers on both the WILDFORCE [4] and the SLAAC1-V [5] boards. The theory behind this type of multipliers was discussed in Section 2.2. There are two types of multipliers implemented; both use Booth's algorithm [6]. The first type, called the regular constant multiplier, uses Booth's algorithm directly on the 1's in the binary representation of the constant. The second type implemented is based on the New Technique that was introduced in Section 2.2, along with Booth's algorithm. The details of both implementations will be provided along with area and speed results collected, comparing both types with the modgen multiplier of JHDL [9]. The modgen multiplier provided with JHDL, is a hand-crafted solution targeted to Xilinx FPGAs.

3.1 Regular Constant Multiplier

The implementation created for this multiplier is able to support different sizes for the exponent and the mantissa. The area and speed testing that will be shown is based on the IEEE format [10] where the exponent is 8 bits wide and the mantissa is 24 bits wide; only 23 bits of the mantissa are recorded because there is an implicit '1' in the most significant position. Thus, the whole data word is 32 bits wide, counting the bit for the sign.

The floating-point multiplication that is used with IEEE format consists of several steps as discussed in Section 2.3. The most time and area consuming step is the multiplication of the mantissas. This multiplication is the primary focus of this section and also where the regular constant multiplier is used. After determining the binary representation of the constant to be used as the coefficient in the multiplier, the process begins by checking the positions of the 1's in the representation. When there is a binary 1 surrounded by two 0's (010), it is counted as one term to be used in the overall addition operation. When there are two consecutive 1's, they are counted as two separate terms. A string of more than two consecutive 1's is dealt with using Booth's algorithm resulting in two terms, one of which is positive and the other negative. With this implementation, any group of 1's will end up being at most two terms in the overall addition that provides the product.

The position of any bit that maps to a term determines the number of shifts that are needed so that the terms line up correctly in the addition step [6]. This is depicted in Figure 5. The terms that are derived from the binary representation are added with an adder tree.

The terms that can exploit the benefits of Booth's algorithm are aligned together. Given the nature of Booth's algorithm, the number of subtractors required is half the number of terms that use Booth's algorithm. The other terms are grouped in pairs and added together. The terms that are produced from the Booth's algorithm terms are always positive because the positive term is always larger than the negative term. Thus, there will not be any subtractors in the adder tree at a level above the leaves. After acquiring the product of the mantissas from the adder tree, normalization takes place and the sum of the exponents is fixed to reflect the effect of the normalization. This process will end in the product of the two floating-point numbers.

The structure of the adder tree offers a very good pipelining mechanism. Registers installed between the outputs of a stage and the inputs of the following stage provide one clock cycle delay of the whole result. The implemented multipliers give the user the option of pipelining, as well as a choice in the number of stages in the pipeline. The fully pipelined case will occur when all the adders have a register at their output. The number of stages of the fully pipelined stage depends on the number of terms to be added. The terms form the leaves of a binary tree; therefore, the depth of the tree will be $\lceil \log_2(NumberOfTerms) \rceil$.

Figure 9 shows an example adder tree of a regular constant multiplier with the constant coefficient set to 183, the binary representation of this number is (10110111). Approaching this representation right to left, the first three 1's make use of Booth's algorithm, the next three 1's end up in three extra terms to be added. The terms are shifted versions of the number that is to be multiplied with the constant coefficient. Figure 9 shows the needed shifts.

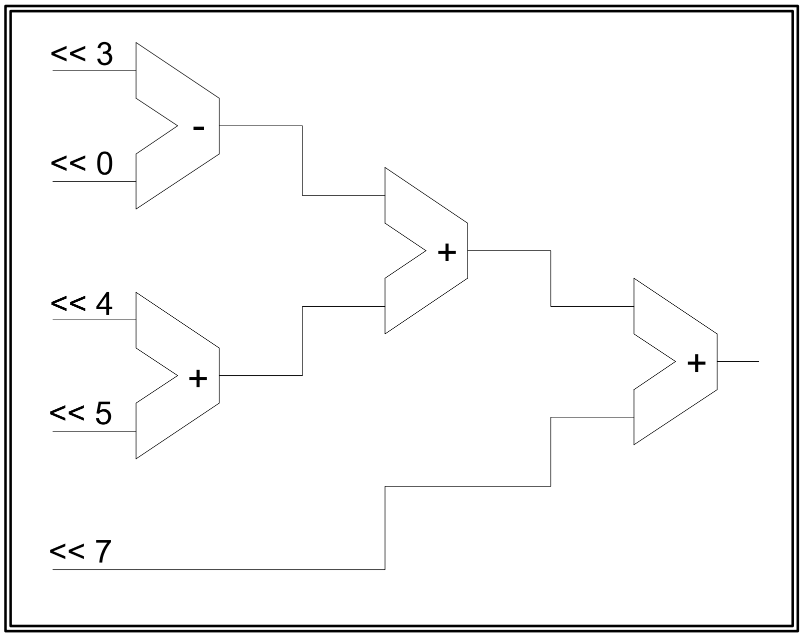


Figure 9 Regular constant multiplier (const = 10110111)

3.2 Conversion Constant Multiplier

The implementation of the conversion constant multiplier offers the capability of specifying the number of bits for the mantissa and the exponent for the floating-point representation. Using the same approach as with the regular constant multipliers, the results of area consumption and speed are reported for the multiplier. This multiplier has 8 bits for the exponent and 24 bits for the mantissa (23 bits are actually saved because of the implicit 1 in the most significant bit), the same as the IEEE format [10].

As discussed above, the most time consuming step in the floating-point multiplication is the multiplication of the mantissas. The following subsection will focus on this multiplication as it is done through the use of the conversion constant multiplier. The discussion to follow is based on the “new technique” which was explained in Section 2.2.

The multiplication is based on the binary representation of the constant coefficient. If the coefficient is being represented with n bits, it can be observed that this coefficient can be recreated by subtracting its 1’s complement from the largest number that can be represented with n bits, which is n 1’s. As an example, consider the coefficient 183 (10110111) used in the previous section. This coefficient requires at least eight bits to be represented, so consider n to be nine bits. The largest number that can be represented is 511 (111111111), and the 1’s complement of 183 (010110111) is 328 (101001000). Then multiplication by the coefficient 183 can be substituted with a multiplication by (511 – 328). Because 511 is all 1’s, the use of Booth’s algorithm will result in two terms. The same approach used with 183 in the regular constant multiplier is used with 328.

The same rules used in the preceding section regarding the use of Booth's Algorithm and the bit-wise shifts are applied here. The first two terms are the two terms created from applying Booth's on the coefficient 511. A tree of adders is now created based on the 1's in the binary representation of 328 (101001000). There are three terms created from this representation based on the four 1's, since there are no three or more consecutive 1's. The result from the tree of adders will be subtracted from the result obtained from the first subtractor that represents the coefficient 511. The overall multiplier will have two subtractors outside the tree of adders, which will be created of two adders that will take care of summing the three terms. Figure 10 shows the structure of this multiplier with the bit-wise shifts needed at the different inputs.

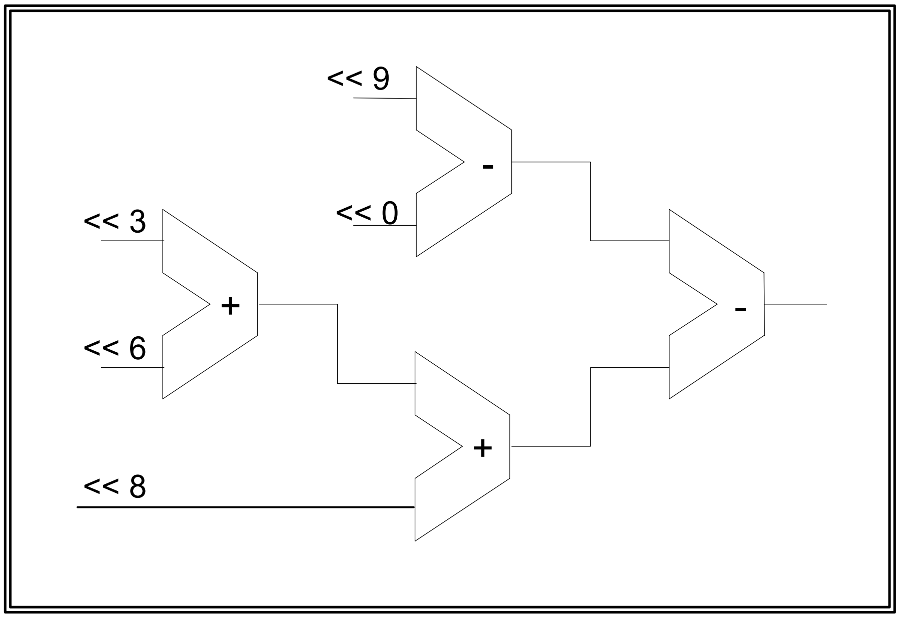


Figure 10 Conversion constant multiplier (const = 10110111)

The structure of the multiplier in this case, as in the regular constant multiplier case, offers a good pipelining mechanism by installing registers between the separate stages. The implemented multiplier in this case offers same pipelining options as the regular multiplier. The fully pipelined version will be where all the outputs of the adders and subtractors have a register. The number of stages in the fully pipelined case depends on the number of terms to be added. With the conversion constant multiplier there is an extra stage added by the two subtractors needed to fulfill the conversion technique. The two subtractors in Figure 10 are the ones created by first implementing Booth's algorithm on the all 1's number and the second subtractor subtracts the result of the adder tree (based on the 1's complement of the coefficient) from the result of the first subtractor. Thus, in this case, the depth of the stages will be $\lceil \log_2(NumberOfTerms) \rceil + 1$, the NumberOfTerms in this case in the number of terms from the adder tree; that is, the number of terms from the 1's complement of the constant coefficient.

3.3 Implementation Statistics

The results collected after synthesizing and running the multipliers on the WILDFORCE and on the SLAAC1-V boards are provided in the following section. Each set of results provided shows the fastest clock speed at which the multiplier runs along with the hardware resources used, as reported by the synthesis tools. Each set compares the different multipliers (regular constant coefficient, conversion constant coefficient, and the JHDL modgen multiplier) to each other, considering different constants. The three sets considered are: unpipelined and pipelined multipliers implemented on the WILDFORCE Board, and unpipelined and pipelined multipliers implemented on the SLAAC1-V board.

3.3.1 Unpipelined Multipliers Implemented on the WILDFORCE Board

Table1 shows the sizes and the maximum clock speed for the unpipelined multipliers using the constants shown.

The terms in the table will be explained in the following section. The XC4000 signifies the Xilinx parts [15] used on the WILDFORCE board on which these multipliers have been implemented. The sign, exponent, and the mantissa show these values of the constants. The mantissa (binary) shows the binary format of the mantissa along with the implicit one in the most significant bit position.

The highest speed column gives the highest clock cycle frequency of the WILDFORCE board for which the multipliers still give no errors in computing the product. The testing mechanism will be discussed later. The next four columns: CLB usage, FLOPS usage, 4-input LUT, and 3-input LUT are reported directly from the Xilinx synthesis tools. The last column, fixed multiplier # of terms, shows the number of terms that the constant coefficient multipliers have to add; this number is important since it reflects the depth of the tree of adders, discussed in Sections 3.1 and 3.2.

To pick the constants to use in these tables, a function was created to estimate the size that would be needed in the implementation of the multiplier. The constant coefficient multipliers of the same type, regular or conversion, follow the same structure; the difference is in the number and the size of the adders needed to create the adder tree. Accounting for these two factors gives an estimate of the size of each multiplier in comparison with the same type of multiplier, but with a different constant. The function created was used on all the constants that can be created from the 23 bits of the mantissa. These numbers were sorted using Matlab [16], the constant creating the maximum size of the regular constant coefficient multipler was picked. Then, from the same sorted array the constants whose sizes were in the ¼, ½, and ¾ positions of the array were picked.

The conversion constant coefficient multipliers require more logic to be created, on average, than the regular constant coefficient multipliers. This extra requirement is due to the use of two extra terms, as discussed in Section 3.3. Due to this observation, the tables were created based on the sizes of the regular constant coefficient multipliers. The same constants are used to create the conversion constant coefficient multipliers. For each constant, the multiplier that uses the least area is utilized. To complete the tables, two extra values were added. The first value shows the maximum area that can be used after choosing the smaller of the two types. This constant is the value whose binary representation has no consecutive ones or zeros, and the representation starts and ends with ones, in the case of an odd number of bits in the mantissa. When the conversion technique is used on this value, the result is a representation of no consecutive ones or zeroes, with zeroes at the boundary. Even that the number of ones is less in the use of the conversion case, but the use of the conversion technique requires extra logic that balances out the decrease in the number of ones. The second value has no ones in the mantissa representation; the value used has 128 as the exponent so that the multiplier multiplies by two.

The JHDL modgen multiplier is a multiplier that has two variable inputs, thus the size of the multiplier should be independent of the values that are multiplied. This is true in the case when the two inputs are left as variables, and the reported values of that case are shown at the bottom of the rows assigned for the JHDL modgen multiplier. In the case where one of the inputs to this multiplier is declared as a constant value, the synthesis tools were able to introduce certain optimizations that reduced the size of the multiplier and increased the speed at which this multiplier could operate.

A function was developed that estimates the size of the constant multipliers for every different constant, using this function the specific coefficients were picked to report the results that are shown below.

The mantissa value of 6141659 is the constant that creates the greatest area consumption, when implemented in hardware, the CLB usage is 397 CLBs, the conversion constant multiplier for the same constant uses 250 CLBs. The JHDL modgen multiplier with two variable inputs uses 368 CLBs, while in the case of a constant input with a mantissa of 6141659, the CLB usage is 352. For these specific numbers, using the conversion constant multiplier is the best choice in saving area. For each constant the smaller between the regular and the conversion constant multipliers should be used to get good saving results.

The maximum operating clock frequency is another important aspect to consider. For the mantissa value of 6141659, the regular constant multiplier can run at 18 MHz, while the conversion constant multiplier can operate at 25 MHz; both these numbers are a significant gain over the operating frequency of the JHDL multiplier which is 9 MHz using two variable inputs and also with 6141659 as the mantissa of a constant input.

The number of terms that are to be added in the constant multipliers can give a good idea about the performance of the regular and the conversion versions of the constant multipliers with respect to each other. The regular constant multiplier has sixteen terms to add while the conversion multiplier has nine terms to add.

Another important constant to consider is the maximum constant multiplier that can be created while choosing the smaller of these two versions. In Table1, the constant that shows this case is 5592405. The area consumption of both versions of the constant multiplier is 330 CLBs (both add 13 terms), while that of the JHDL modgen with the same constant input is 344 CLBs. The savings in area consumption in this case are not significant. The highest speed at which this multiplier can run is 23 MHz using the regular version, while it is 22 MHz using the conversion type. The JHDL modgen runs at a maximum of 10 MHz.

The numbers 6321838, 4255289, and 2097828 map respectively to the values that were picked to represent the ¾, ½, and ¼ positions in the sorted array of the estimated mulitplier sizes. Using the smaller of the two versions in each case still saves on the area used by the JHDL modgen multiplier; there is also a significant increase in the speed at which these multipliers can run.

The constant that maps to multiplying by two, shows that the JHDL modgen multiplier uses the least area, 44 CLBs, while the area used is 54 CLBs for the regular constant multiplier and 149 CLBs for the conversion constant multiplier version. The WILDFORCE board can run the clock at a maximum of 50 MHz; both the regular and the JHDL modgen multipliers run at that speed, while the conversion type runs at 33 MHz.

3.3.2 Pipelined Multipliers Implemented on the WILDFORCE Board

An important feature of these constant coefficient multipliers and of the JHDL modgen multiplier is the ability to be pipelined. Pipelining helps by increasing the value of the highest rate at which the clock can be run [17]. Introducing delays between the stages of the adder tree pipelines the constant coefficient multipliers, thus useful pipelining is limited by the depth of the adder tree. The modgen JHDL multiplier offers a greater number of pipelining stages. To compare the different multipliers, the regular and the conversion constant multipliers are fully pipelined, and the number of stages created is reported. The JHDL multiplier is then pipelined the number of stages that is typically used in both the previous multipliers. The results for the pipelined multipliers are reported in Table 2.

To create the pipelined version of the constant multipliers, the modgen adders of JHDL were used. The outputs of these adders can be registered without extra area cost. The limitation of these adders is that they only support 32-bit inputs and output. This problem was fixed by truncating every result in the adder tree to 32 bits; the lost accuracy is insignificant as will be discussed later.

The extra column in Table 2 shows the number of pipelined stages. As can be seen from this table, the number of stages typically used is 4, this is the number of stages used in the JHDL modgen multiplier.

In this implementation, the regular constant coefficient multiplier did not break at the speeds that can run on the WILDFORCE board. The conversion constant coefficient multiplier ran at 36 MHz in its worst case. When picking the smaller type for each constant the constant multipliers do not break on the WILDFORCE board. The JHDL multiplier with four pipelined stages did not break, only in the case when multiplying with the constant 2. The other cases operated between 32 and 40 MHz. When the JHDL multiplier was implemented with the two inputs variable, it operated at a maximum clock rate of 27 MHz. In the reported cases above, the constant multipliers performed better for all case.

X 4000	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	CLB Usage	FLOPS Usage	4-input LUT	3-input LUT	Fixed Multiplier # Of Terms
	0	127	6141659	1.10111011011011011011011	18	397	2	647	25	16
	0	127	5592405	1.10101010101010101010101	23	330	2	521	25	13
	0	127	6321838	1.11000000111011010101110	20	269	2	404	25	10
Regular	0	127	4255289	1.10000001110111000111001	25	252	2	372	25	9
	0	127	2097828	1.01000000000001010100100	35	178	2	231	25	6
	0	128	0	0	>= 50	54	2	54	3	1
	0	127	6141659	1.10111011011011011011011	25	250	2	367	25	9
	0	127	5592405	1.10101010101010101010101	22	330	2	518	25	13
	0	127	6321838	1.11000000111011010101110	27	250	2	365	25	9
Conversion	0	127	4255289	1.10000001110111000111001	26	251	2	367	25	9
	0	127	2097828	1.01000000000001010100100	19	288	2	437	25	11
	0	128	0	0	33	149	2	176	25	4
	0	127	6141659	1.10111011011011011011011	9	352	2	618	1	
	0	127	5592405	1.10101010101010101010101	10	344	2	610	3	
	0	127	6321838	1.11000000111011010101110	11	317	2	558	1	
Jhdl	0	127	4255289	1.10000001110111000111001	9	323	2	570	1	
	0	127	2097828	1.01000000000001010100100	16	250	2	440	3	
	0	128	0	0	>= 50	44	2	80	3	
	Two variables, no constants				9	368	2	635	7	

Table 1 Results of the unpipelined multipliers implemented on the WILDFORCE board

X 4000 (Pipelined)	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	CLB Usage	FLOPS Usage	4-input LUT	3-input LUT	Pipeline Stages
	0	127	6141659	1.10111011011011011011011	>= 50	344	407	475	26	4
	0	127	5592405	1.10101010101010101010101	>= 50	322	396	427	48	4
	0	127	6321838	1.11000000111011010101110	>= 50	261	267	296	40	4
Regular	0	127	4255289	1.10000001110111000111001	>= 50	269	307	295	59	4
	0	127	2097828	1.01000000000001010100100	>= 50	195	180	211	41	3
	0	128	0	0	>= 50	54	2	54	3	0
	0	127	6141659	1.10111011011011011011011	>= 50	285	237	309	48	4
	0	127	5592405	1.10101010101010101010101	36	368	385	430	63	5
	0	127	6321838	1.11000000111011010101110	>= 50	285	228	299	48	4
Conversion	0	127	4255289	1.10000001110111000111001	44	282	226	296	49	4
	0	127	2097828	1.01000000000001010100100	>= 50	339	330	355	73	5
	0	128	0	0	>= 50	181	106	162	52	2
	0	127	6141659	1.10111011011011011011011	32	376	216	645	43	4
	0	127	5592405	1.10101010101010101010101	39	372	213	652	43	4
	0	127	6321838	1.11000000111011010101110	39	362	212	621	42	4
Jhdl	0	127	4255289	1.10000001110111000111001	39	371	210	635	42	4
	0	127	2097828	1.01000000000001010100100	40	342	207	589	42	4
	0	128	0	0	>= 50	239	181	365	40	4
	Two variables, no constants				27	386	223	657	51	4

Table 2 Results of the pipelined multipliers implemented on the WILDFORCE board

3.3.3 Unpipelined Multipliers Implemented on the SLAAC1-V Board

The multipliers implemented on the WILDFORCE board gave a very good result after being pipelined. The next inquiry is, how these multipliers will perform on a Virtex chip. The unpipelined version of both the constant coefficient multipliers and of the JHDL modgen multiplier were implemented and tested on the SLAAC1-V board. This board uses the Virtex 1000 Xilinx chips as its Processing Elements (PEs). [5]

Table 3 shows the results obtained; they are a bit different from the data reported for the WILDFORCE board. The area consumption in this case is measured in slices as opposed to CLBs in the case of the XC4000 PEs. The other hardware consumption data reported is the use of the 4-input LUTs, with the WILDFORCE there was also 3-input LUTs.

The relationship between the area consumption results with respect to the different types of the multipliers is quite close to those reported in the case of the WILDFORCE board. The maximum slice usage of the regular constant coefficient multiplier is 327 for the 6141659 constant; the corresponding conversion multiplier size is 198, while the JHDL multiplier used 305 slices. The largest multiplier that can be created while choosing between the two constant coefficient multipliers (constant = 5592405) uses 253 slices. The JHDL modgen multiplier uses 314 slices when both inputs are variable. When picking the smaller of the two types of constant multipliers, better area consumption in given in all reported cases.

The real significant performance enhancement is in the highest clock speed at which the constant multipliers can run. The lowest speed reported on the constant multipliers, while choosing the multiplier with the lower area consumption, is 52 MHz. The JHDL multiplier did not provide such a good result; it ran at 14 MHz when both inputs were variable and between 15 and 18 MHz for the other cases, excluding the case when multiplying by two where this multiplier did not break.

3.3.4 Pipelined Multipliers Implemented on the SLAAC1-V Board

The last implementation that will be discussed is that of the pipelined multipliers on the SLAAC1-V board. The area consumption in this case of the constant multipliers is larger than that of the JHDL multipliers because the modgen adders were not supported for the Virtex chips and thus regular adders were used. The registers used for pipelining were mapped to their own slices (without the use of placement tools) and thus more area was consumed. Table 4 shows the results collected, and follows directly from Table 3 on format it also follows the same discussion in Section 3.3.2 on the number of pipelined stages to use with the modgen multiplier.

The results of the area consumption can be compared between the two types of constant multipliers. The largest regular constant multiplier (constant = 6141659) used 848 slices, while the same constant needed 629 slices in the conversion constant multiplier. The largest multiplier that can be created while picking the smaller of the two types (constant = 5592405), consumes 748 slices with the regular multiplier and 835 with the conversion multiplier. The size of the JHDL modgen when both the inputs are variable is 411 slices. It can be seen that the registers did require a lot of space.

The fastest clock speed results still show that the constant multipliers can work at higher speeds. All coefficients used with the regular constant multiplier resulted in logic that did not break at the speeds at which the SLAAC1-V board can run. The conversion constant multiplier ran at the lowest speed of 80 MHz for the largest multiplier considered. The modgen multiplier had speeds between 57 and 61 MHz, excluding the case of multiplying by 2. The constant multipliers still provided operation at higher speeds even though they used more area on the chip.

The results reported in the four tables show the improvement in area consumption and in high clock speed operation that the constant multipliers offer. This improvement is of use in the case where the multipliers in a design have a fixed coefficient or when the coefficients are constant for a long time.

Virtex	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	Slice Usage	4-input LUT	Fixed Multiplier # Of Terms
Regular	0	127	6141659	1.10111011011011011011011	42	327	502	16
	0	127	5592405	1.10101010101010101010101	50	252	405	13
	0	127	6321838	1.11000000111011010101110	48	242	354	10
	0	127	4255289	1.10000001110111000111001	51	231	321	9
	0	127	2097828	1.01000000000001010100100	68	140	216	6
	0	128	0	0	>= 147	56	87	1
Conversion	0	127	6141659	1.10111011011011011011011	53	198	336	9
	0	127	5592405	1.10101010101010101010101	48	253	435	13
	0	127	6321838	1.11000000111011010101110	55	223	359	9
	0	127	4255289	1.10000001110111000111001	52	229	368	9
	0	127	2097828	1.01000000000001010100100	43	260	413	11
	0	128	0	0	87	140	253	4
Jhdl	0	127	6141659	1.10111011011011011011011	15	305	430	
	0	127	5592405	1.10101010101010101010101	15	305	334	
	0	127	6321838	1.11000000111011010101110	15	293	334	
	0	127	4255289	1.10000001110111000111001	16	305	310	
	0	127	2097828	1.01000000000001010100100	18	281	166	
	0	128	0	0	>=147	195	63	
	Two variables, no constants				14	314	618	

Table 3 Results of the unpipelined multipliers implemented on the SLAAC1-V board

Virtex (Pipelined)	Sign	Exponent	Mantissa	Mantissa (Binary)	Highest Speed (MHz)	Slice Usage	4-input LUT	Pipeline Stages	Fixed Multiplier # Of Terms
Regular	0	127	6141659	1.10111011011011011011011	>=	848	502	4	16
	0	127	5592405	1.10101010101010101010101	>=	748	405	4	13
	0	127	6321838	1.11000000111011010101110	>=	642	354	4	10
	0	127	4255289	1.10000001110111000111001	>=	615	321	4	9
	0	127	2097828	1.01000000000001010100100	>=	448	216	3	6
	0	128	0	0	>= 147	200	87	0	1
Conversion	0	127	6141659	1.10111011011011011011011	87	629	336	4	9
	0	127	5592405	1.10101010101010101010101	80	835	435	5	13
	0	127	6321838	1.11000000111011010101110	84	628	359	4	9
	0	127	4255289	1.10000001110111000111001	84	629	368	4	9
	0	127	2097828	1.01000000000001010100100	87	755	413	5	11
	0	128	0	0	>=	362	253	2	4
Jhdl	0	127	6141659	1.10111011011011011011011	57	394	430	4	
	0	127	5592405	1.10101010101010101010101	58	390	334	4	
	0	127	6321838	1.11000000111011010101110	62	375	334	4	
	0	127	4255289	1.10000001110111000111001	61	384	310	4	
	0	127	2097828	1.01000000000001010100100	61	355	166	4	
	0	128	0	0	>=	325	63	4	
	Two variables, no constants				58	411	618	4	

Table 4 Results of the pipelined multipliers implemented on the SLAAC1-V board

Chapter 4: FIR Filter Implementation

This section describes in detail the FIR filter implementations that were considered, namely the direct-form and the lattice structures [3]. The two implementations mentioned above are reconfigurable. The reconfiguration was accomplished in two ways, through the data path and through re-synthesis and full reconfiguration of the PEs. Reconfiguration using the data path is real-time, where the new coefficients that are to be used in the taps of the filter are introduced into the data path. The tap records the new value and uses it as the multiplying coefficient. The multipliers used are the regular floating-point multipliers provided in the modgen package with JHDL. The drawback to these multipliers is that they are generic, and thus cannot take advantage of the fact that the coefficient in each tap may be fixed for a long time. If the coefficients do not change frequently, then the constant coefficient multipliers discussed in the previous chapter are well suited for the job. These multipliers offer enhancements in computational speeds and in chip area consumption. These multipliers have to be re-synthesized for each new coefficient. Thus, there will be time loss with respect to the generic multipliers when changing the response of the filter. The following subsections will discuss the implementation of each FIR format, irrespective of the multipliers used, and the processes by which to reconfigure the response of the filter when using the different multipliers. A final subsection will discuss the relative merits of each of the four resulting configurations of the FIR filter.

4.1 Direct-Form FIR Filter

As discussed in Section 2.1, the direct-form FIR form follows directly from the non-recursive equation (1) governing the response of the filter. Figure1 shows the structure of this filter. The discussion in the following subsection deals with the implementation of the direct-form, irrespective of the multipliers used, after describing the mechanism by which the constants of the multipliers are set in the case of data-path reconfiguration.

The data path that traverses PE1 through PE4 through the systolic bus on the WILDFORCE board was chosen for the hardware implementation on the FIR filter. Each PE can hold one or two taps, which will relay data to each other through the systolic bus. PE1 will get the input data from FIFO1; after processing this data, it will pass it to PE2. PE4 will get the data after it is processed in the data path; and then outputs it to FIFO4. With this structure, the host code provides data to FIFO1 to be processed and then retrieves the processed data from FIFO4.

Each tap in this structure needs two inputs and two outputs as shown in Figure 11. Time multiplexing was used so that each path will use every other clock cycle on the systolic data bus because the PEs have only one input and one output to the bus.

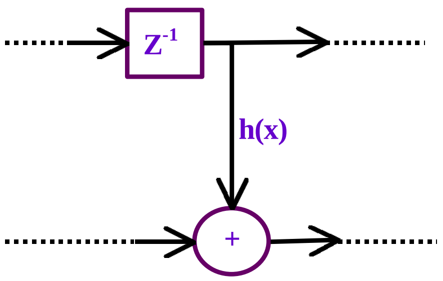


Figure 11 A single tap of the direct-form FIR filter [3]

Control bits were introduced at the top four bits of each word passing through the filter. These bits are used to differentiate between the different control and data words that will pass through the PEs. There are two types of valid data words which map to the two paths that need to be multiplexed into one path. The other control bits are used to either reset the different FSMs (Finite State Machines) used, declare that the words going in are valid or not, or to setup the coefficients in the case of data path reconfiguration. Table1 shows the different combinations of these control bits and what meaning they encode.

Control Bits	30	29	28	Description
	V	R	P	
	0	0	0	Invalid Data Mode
	0	0	1	Invalid Program Mode
	0	1	0	
	0	1	1	Reset Signal
	1	0	0	Valid Mode (Original Data)
	1	0	1	Valid Program Mode
	1	1	0	Valid Data Mode (Accumulation Data)
	1	1	1	Reset Signal

Table 5 Control bits used in the filter design

Bit 31 is tied directly to the *fifoempty* flag of PE1which is considered as another valid bit. Thus, whenever the FIFO is empty the words are considered invalid. This bit is also tied to the *fifoalmostfull* flag of PE4; this flag is sent back from PE4 to PE1 through the crossbar and gives the control needed so that no data is overwritten in PE4's FIFO. The bits that are shown in Table1 are V-valid, R-Reset, and P-Program.

Figure 12 shows the structure implemented on the PEs in the case of data-path reconfiguration. RegC takes care of saving the value of the coefficient for the multiplier and in resetting this value when needed. When the values inside the RegCs of the different taps are to be changed, the Reset word is first sent through the board. This signal will reset the FSM (Finite State Machine) inside RegC to the initial case. The RegCs now start saving the new coefficients that are passed to them, each RegC uses the first valid coefficient it gets and then invalidates it. Thus, the first RegC will get the first coefficient and so on.

Figure 13 shows the same implementation using the constant coefficient multipliers, so there is no need for RegC to hold the coefficient.

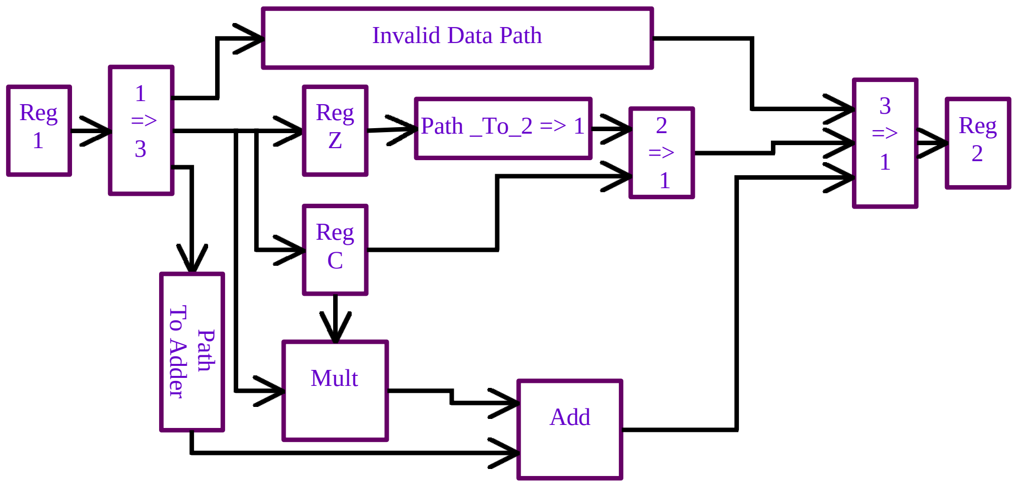


Figure 12 Implemented structure of the direct-form FIR filter with regular multipliers

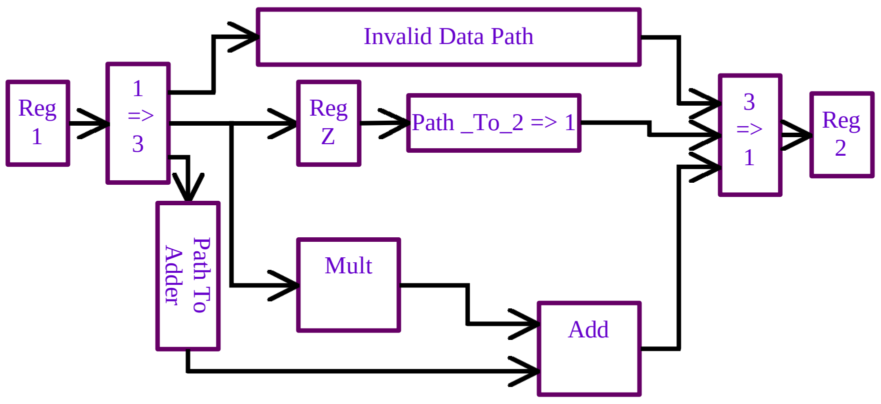


Figure 13 Implemented structure of the direct-form FIR Filter with constant coefficient multpiliers

The valid data mode words that go through the filter should always be passed as Original data (from Table1) first followed by Accumulation data. This order is important so that the intermediate accumulation result is attributed to the correct Original data. The "1 => 3" and the "3 => 1" modules are responsible for keeping the order of the two types of valid data.

4.2 Lattice FIR Filter Structure

The response of the lattice structure of the FIR filter was discussed in Chapter 2. Equations (6 & 7) describe the response of the first tap of the lattice structure, Equations (8 & 9) for the second stage follow in the same manner, with the following taps following a similar pattern. The hardware implementation of this type of filters is discussed next. Using the same approach as Section 4.1, the discussion will focus on the overall implementation, irrespective of the multipliers used, and then the differences between the two approaches will be pointed out.

The process used in setting the coefficients when using the regular multipliers is the same that was used with the direct-form implementation. The same data path used in the direct-form filter is used with the lattice structure. The input data to the filter comes through FIFO1 to PE1, then the data traverses the systolic bus through the four PEs and the data is sent to the host program through FIFO4. In this structure each PE can hold an arbitrary number of taps of the FIR filter, as long as the design still fits on the chip.

From Figures 2 and 3 it can be seen that there are two input and two output paths on each tap of the lattice structure; the same approach in the direct-form is used, where the two paths are multiplexed into one and the control bits are used to distinguish between them. The two paths are the “f” data and the “g” data that are shown in Figures 2 and 3 [3], thus each of these paths will use every other clock cycle on the systolic bus.

The control bits are used to distinguish the different data types and indicate the validity of the data. These bits identify control words that deal with setting the coefficients when using the JHDL multipliers and resetting the filter, and the two types of data words that are used to represent the “f” and the “g” paths. The control bits are also used to declare if the words are valid or not. The following table shows the different combinations of these bits and their encoded meaning.

Control Bits	30	29	28	Description
	V	R	P	
	0	0	0	Invalid Data Mode
	0	0	1	Invalid Program Mode
	0	1	0	
	0	1	1	Reset Signal
	1	0	0	Valid Mode ("f" Data)
	1	0	1	Valid Program Mode
	1	1	0	Valid Data Mode ("g" Data)
	1	1	1	Reset Signal

Table 6 Control bits used in the lattice structure FIR filter

As in the direct-form implementation, bit 31 is used to also declare if the values are valid or not by being tied directly to the *fifoempty* flag of FIFO1. When the FIFO is empty, the data being read is invalid. This bit is also tied to the *fifoalmostfull* flag of PE4 through the crossbar; this signal is needed so that no data is overwritten inside FIFO4. Also following from the previous structure, the bits are V for the valid control signal, R for the Reset signal, and P for distinguishing between program data and regular data.

The forms of lattice structure implementation with the regular multipliers and the constant coefficient multipliers are shown in Figures 14 and 15 respectively. The difference comes from the hardware that was used to store the coefficients for the JHDL multipliers.

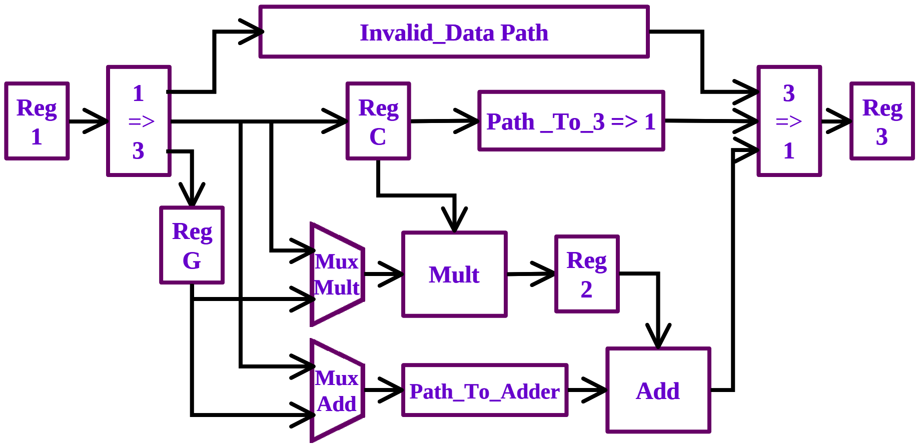


Figure 14 Implemented structure for the lattice structure with regular multipliers

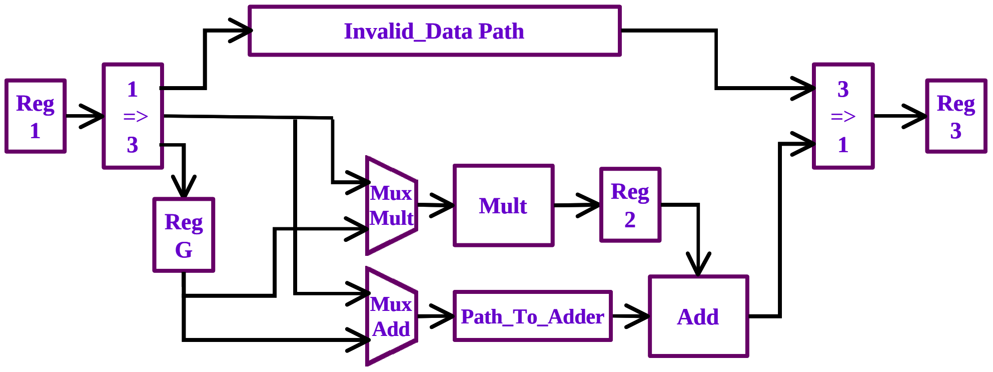


Figure 15 Implemented structure for the lattice structure with constant coefficient multipliers

From the lattice structure of the FIR filter that is shown in Figures 2 and 3, we see that both the “f” and the “g” path are multiplied by the same coefficient. In the implementation that is used on the PEs, the two data paths “f” and “g” will be multiplexed. With the correct control of the data that is entering the PE, the “f” and “g” values can be multiplied and added using the same multiplier and adder, thus saving a significant amount of area on the chip.

The individual modules in this implementation follow exactly from their counterparts in the direct-form structure except for the RegG, Mux Mult, and Mux Add. The “Mux Mult” and the “Mux Add” are two multiplexers that take care of sending the “f” and “g” data at the right time to the multiplier and adder, so that the output is the same as the one expected of the lattice-structure, while using only one multiplier and one adder.

The “f” and the “g” data should be kept in the correct sequence, in the same manner that the Original and the Accumulated data had to be kept in sequence in the direct-form implementation.

4.3 Implementation Results

There are four different possible implementations that can be created of the FIR filter. These four implementations rise from first choosing which form to use, the direct-form or the lattice structure. The second step is picking either the JHDL modgen multipliers or the constant coefficient multipliers. Another parameter to consider is the number of filter taps in each PE.

When using the JHDL modgen multipliers, the two inputs to the multipliers are variable. This helps the FIR implementation (both the direct-form and the lattice structure) to be real-time reconfigurable. The process of this reconfiguration was discussed in Subsection 4.1 and 4.2, the reconfiguration process needs to send a reset signal followed by the coefficients that are to be used as the new constants of the filter. The time needed for that depends on the depth of the filter and on the number of pipelined stages used, thus it is the time needed for the values to pass through the logic of the filter. Another observation is that the new values to be filtered by the new filter configuration need not be delayed after the coefficients are sent, so the delay is only the number of the values sent to reconfigure (each value needs one clock cycle to get to the input of the filter).

The constant multipliers offer faster operating clock rates and smaller area, but the run-time reconfigurability is out of the question. To change the response of the filter the whole multipliers should be changed, synthesis will be needed after the code for the multipliers is changed. The process of changing the coefficients and resynthesizing is time consuming; the synthesis of four PEs for one board can take up to an hour with the conventional methods. The use of JBits [18] can help in this time consumption, the reconfiguration of one PE on the WILDFORCE will approximately require 190 ms, changing the response for the three PEs would take 3 * 190 ms. Throughput for each period T that includes a response change will be (T – 3*190) * maximum clock speed. When using the JHDL multipliers for the same time period, setting the response only takes three clock cycles which is negligible, so the throughput will be T* maximum clock speed with the JHDL multipliers. The larger throughput for each case of time of operation and maximum clock speed determines which multipliers to use with the FIR filter needed.

The JHDL modgen multipliers used in the FIR filter have two variable inputs and thus the size of these multipliers is not dependent on the values at the inputs. The area consumption of these filters and the maximum operating clock speed are reported in Table 7.

	One Tap Per PE		Two Taps Per PE		Four Taps Per PE
	Area Consumption	Clock Speed	Area Consumption	Clock Speed	Area Consumption
Direct-Form JHDL Multipliers	1097	24	2231	20	4226
Direct-Form Constant Multipliers	1055	27	2119	20	4083
Lattice Structure JHDL Multipliers	1104	20	2207	17	4224
Lattice Structure Constant Multipliers	1037	24	2074	18	4096

Table 7 Implementation results for the different FIR filters

Reporting the area consumption of the FIR filters using the constant multipliers is more complex. The size of the multipliers is dependent on the constant to be used for the multiplication. Another issue is that for each constant there are two versions of the constant multiplier (regular and conversion); they give the same result, but one version needs less area for each different constant. The reported values will be based on the constant that creates the average size of the minimum of both types of constant multipliers. After determining the sizes needed for each constant for both types, one of the values that create the median of the minimum is 26810 (1.0000110100010111010). After using this value in the implementation of the FIR filters, the area consumption results are reported in Table 7.

In all implementations of the FIR filter, the JHDL multipliers were pipelined at 10 stages and the constant multipliers were fully pipelined. Then, an extra delay was added to get to the 10 stages of the JHDL multipliers (this delay was not added while testing for area). With this setup, the speed at which the filters break is not limited by the multipliers. Table 7 reports the maximum speeds at which each of the filter implementations was able to run.

The implementations using the constant multipliers have a better clock speed performance since there is less logic in the implementation. The performance of the multipliers was of no effect because the multipliers were shown to work at higher speeds with no errors, while being pipelined with fewer stages, results of Subsection 3.3.

The constant multipliers proved to use less area and to run at higher speeds, reported results in Subsection 3.3. When these multipliers replace the JHDL modgen multipliers on the FIR filters, the area savings are not significant because of all the logic in the design.

Chapter 5: Fast FIFO

Data access on the WILDFORCE board can be accomplished by the use of the hardware FIFOs on PE0, PE1, and PE4 [4]. These FIFOs offer a method to get data on and off the board with flow-control. Consuming the CPU and having a slow data transfer rate are the two primary problems of the FIFOs on the WILDFORCE board. The WILDFORCE board offers the ability to access the memories of the PEs from the host program with and without DMA (Direct Memory Access). Using the memory as the main data interaction point adds to the complexity of the hardware, because addressing and memory bus handling must be integrated into the FPGA design. The Fast FIFO was developed to offer the same simple interface as the regular FIFOs along with the speed that can be attained by using DMA to the memories. The following sections will discuss the implementation and the results obtained from running the Fast FIFO [4].

5.1 Fast FIFO Concept

Data access to the memories of the PEs is faster than data access to the FIFOs, this fact is the main motivation behind creating the Fast FIFOs. To improve the speed that can be achieved on the data path from FIFO1 through the systolic bus to FIFO 4; the Fast FIFO uses the memory of PE0 as the primary temporary storage of data that is to be passed through the systolic bus. PE0's job is to keep track of the size of the data chunks written to its memory and their position. When PE1 asks for data by asserting a control bit, PE0 supplies data through the crossbar. PE1 gets data as if it were coming from a regular FIFO; it does not have to worry about addressing or memory access. The data is processed in PE1, PE2, and PE3. PE4 in this case has to temporarily store the data in the area that it has (inside the PE and not in PE4’s memory). When the storage space is full, PE4 stops PE0 from sending more data, and it moves the data stored to the memory and informs the host program of the size of the data in the memory.

5.2 The Host Program

The host program controls sending data to the board and retrieving it after it has been processed in the PEs. Passing data through the board using the Fast FIFO is done in blocks of 0 to 256 words. The host program moves the blocks of data through DMA access to the memory of PE0. The host code sends a “control” word to the regular FIFOs (FIFO 1) to notify PE0 of the new data block. The regular FIFOs were picked because the PE can check the *fifoempty* flag and determine if any new data is present. The “control” word that is written carries information about the size of the block and its position in PE0’s memory. Each “control” word gives information about one block of data.

Retrieving the data is done in a similar manner. When PE4 has transferred a block to its memory, it notifies the host program by sending a regular FIFO word that contains the size of the block that has been written. PE4 writes the blocks to its memory consecutively, thus the host program should keep track of where it read last from PE4’s memory. The number of words in FIFO 4 reflects the number of blocks in PE4’s memory. With this information, the host program will be able to retrieve all processed data.

5.3 PE0 Implementation

The job of PE0 is to read the data written to its memory by the host program. The whole process to be done in PE0 is controlled by a Finite State Machine (FSM). PE0 checks the *fifoempty* flag of its regular FIFO, looking for new blocks written to its memory. Every word in the regular FIFO maps to a block that was placed in PE0’s memory in a location and of the size shown in the FIFO word itself.

When PE0 finds that there is data for it to retrieve, it reads the amount of data written at the specified location. This data will be pushed to PE1 through the crossbar; PE1 has the capability to ask for data, thus if PE0 does not have an asserted flag that PE1 needs data, it will stop pushing data and stop reading from its memory.

The first job of the FSM is to read the “control” word from the regular FIFO. The size of the block is extracted from the control word and sent to PE4. This information is needed to control the temporary storage (to be discussed in the following subsection). PE0 can then start reading from the memory; there are two branches here, PE0 either has to read a single word or more than that. If a single word is to be read, it is read and pushed to PE1. PE0 waits to be “stalled” by PE4, announcing that the temporary storage in PE4 is full. PE0 waits again to be un-stalled, signaling that PE4 has finished writing from the temporary storage to its memory. The cycle starts again with PE0 checking the regular FIFO for new words. If PE0 is to read more than one word from memory, the basic process is the same, except that PE0 is waiting for multiple words instead of a single word and thus different clock cycles are to be controlled.

5.4 PE4 Implementation

PE4 is responsible for grabbing the data that is passed to it from PE3, and then passing this data to memory and informing the host program of the size of the data that was received.

The process that takes place in PE4 is controlled by two FSMs. The first step in the process is recording the size of the block to be sent. PE0 sends the size of the block to be read from its memory; PE4 stores this word. The implementation in PE4 contains a RAM that can hold up to 256 words for temporary storage. Using this RAM, PE4 starts storing the data that it receives through the data path (usually through PE3). This process is controlled by the first FSM (Write_1), which takes care of writing the correct amount of data into the temporary storage. After all data specified by PE0 are received by PE4, the FSM sends a Stall signal to PE0 that all data has been stored. This FSM signals the second FSM (Write_2) to start its work. Write_2 is responsible for sending the data in temporary storage to the memory of PE4. When all data is transferred, PE4 writes a word into the regular FIFO that contains information about the size of the block that is written in the memory. In this way, the host program knows that new data has been written, and can retrieve it. After the writing phase in PE4 is finished, it sends an “unstall” signal to PE0, so that the cycle can start again.

5.5 Testing Results

The Fast FIFO was compared to the regular FIFOs by sending the same amount of data through each and timing the process. In the case of the Fast FIFO, the host code dumped 256 blocks of 256 words each into the memory of PE0 and signaled PE0 by sending control words to the regular FIFOs. The host code read the same values back from PE4's memory after reading the control words from FIFO4; this process was repeated 10,000 times to decrease the significance of the error caused by the granularity of the timers. The same amount of data was passed through the regular FIFOs; the host code dumped the data into FIFO1 in chunks of 512 words each and then retrieved the data from FIFO4.

Timing of these two transfers is done in two ways. The first method used was timing all the 10,000 runs so that the entire transfer was tested. The reading function is not called until FIFO 4 was almost full to save on time because reading and writing to the regular FIFOs were blocking calls.

Overall Timing	Time Required (sec)	Throuput (Mbytes/s)
Regular FIFOs	362.1673	6.90288
Fast FIFOs	118.5461	21.0888

Table 8 Timing results for data passing through the FIFOs

Processor Timing	Overall Time	Processor Time	Percentage
Regular FIFOs	394.7339	376.117	95.28%
Fast FIFOs	120.6576	47.71392	39.54%

Table 9 Processor time consumption by the FIFOs

Reading and writing to the FIFOs is done through the DMA FIFO read and write, so that less time is needed, as opposed to using the regular FIFO read and write. The results of the timings are reported in Table 8. the Fast FIFO has a throughput 3.055 times larger. The second timing method is aimed at testing the amount of work the processor is using in the actual reading and writing operations as opposed to the total time elapsed during the process. To accomplish this timing, only the write and read functions are timed when they are called. The extra functions needed in timing slow the whole process by a fraction, thus the overall timing and the timing of the reading and writing will be reported in Table 9. The reported percentages show that using the regular FIFOs will render the processor unable to do other tasks, because most of the operating time is consumed by writing and reading. In the case of the Fast FIFO, the write and read functions require a much smaller fraction of the overall time and thus the processor will be able to have other tasks running while data is passed through the board.

Chapter 6: Integration and Testing Results

Developing a fast, scalable, and reconfigurable FIR filter implementation is the goal of this thesis. This implementation can be created through the integration of the various techniques mentioned in the previous chapters.

The different choices available for the FIR implementation create four different filter configurations. The filter can be either direct-form or lattice structure and it can use either the JHDL modgen multipliers or the constant coefficient multipliers. It can also have a tap or two on each PE of the WILDFORCE board (or more on new, larger FPGAs). The use of the constant multipliers creates a simpler logic implementation, as well as a higher operating frequency than the modgen multipliers.

The data input to the filter can be provided by the regular FIFOs that are provided by the WILDFORCE board, or through the use of the Fast FIFO. Operating the regular FIFOs in the DMA mode, and having only two registers in each PE to pass the data from the input to the output, had a transfer rate of 6.90288 Mbytes/sec as reported in subsection 5.5. The same implementation used with the Fast FIFO, had a transfer rate of 21.0888 Mbytes/sec. Thus, having the Fast FIFO provide the input to the filter can improve the speed of the entire operation.

The direct-form FIR filter was implemented with the constant coefficient multipliers and run under both the regular FIFOs and the Fast FIFO. The throughput through the use of the regular FIFOs was 5.89658 Mbytes/sec and through using the Fast FIFOs it was 19.287 Mbytes/sec. We can see that there is a loss of data rate from what was reported before. This decrease is attributed to the pipeline stages in the adder and multiplier; these stages will use clock cycles to fill up before output data can be collected.

To create the best-case scenario, the Fast FIFO was used with the direct-form FIR filter with constant multipliers while implementing two taps in every PE. The throughput of this implementation is 17.865 Mbytes/s. Every two output words represent one output of the FIR filter (Original and Accumulation data). Each output of the FIR filter requires a multiplication and an addition, thus two floating-point operations in every tap. There are two floating-point operations every two clock cycles, or this can be reported as one floating-point operation every clock cycle. The throughput in words/sec is 17.865 Mbytes/s / 4 = 4.46625 Mwords/s. Each words goes through six taps, thus there are six floating point operations on each word every clock cycle; the overall output is 26.7975 MFLOPS.

The previous paragraphs showed how the throughput of the filter can be increased by using the constant multipliers and through the use of the Fast FIFO to input data and collect results at the output of the filter. The scalability issue is addressed by running the filters on multiple boards. The ACS_API is used to load the FIR filter implementation on several boards, and then to “tie up” these boards together to form a larger filter. The ACS_API can make use of the regular FIFOs and the Fast FIFO as means of input and output from the specific boards, then it relays the data between the boards.

	Throughput	
Number of Boards	Mbytes/sec	MFLOPS
1 (without ACS_API)	17.865	26.7975
1	6.838	10.257
4	0.2798	1.6792
7	0.113	1.1866

Table 10 Results of implementation on multiple boards

A decrease in the total throughput of the system is observed with Table 10. Sub-optimal performance as measured by the total throughput of the system is observed for implementations on multiple boards. The abstraction layers of the ACS_API and the communication channels between the boards are the main source of this overhead. The ACS_API is still under construction; proper tuning of the implementation of the communication channels would significantly reduce the communication overhead. The best-case projected results of such an implementation are shown in Table 11.

	Throughput	
Number of Boards	Mbytes/sec	MFLOPS
1 (without ACS_API)	17.865	26.7975
1	6.838	10.257
4	6.838	41.028
7	6.838	71.799

Table 11 Results of the optimistic implementation on multiple boards

The SLAAC1-V board with the three Virtex chips offers a greater amount of resources and speed of operation as opposed to the WILDFORCE board. An implementation of the FIR filter on such a board would result in a much larger MFLOP count because of the faster operation and the extra number of taps that can be implemented in each PE. There are 2304 CLBs on the X400 chips on the WILDFORCE board, while there are 12,288 slices on the Virtex 1000 chips on the SLAAC1-V;

approximately the Virtex chips can hold designs six times larger than the X4000 chips; then 12 taps of the FIR filter should fit on one chip. If the WILDFORCE boards in the Tower Of Power were to be substituted with SLAAC1-V boards will make implementing a 576 (16 boards * 3 chips * 12 taps) tap FIR filter possible.

In the best-case operation, the PCI bus can be considered the limiting factor to the clock speed that the board can be used at. The SLAAC1-V has a 72-bit bus that runs at 66 MHz, this bus takes care of input and output, then this bus can provide 32 bits of input data at every clock cycle. Every PE can hold 12 taps, so there are 12 floating-point operations every clock cycle in each PE; this amounts to 792M floating-point operations (66M*12) on every PE per second. The SLAAC1-V holds three PEs; one of these PEs (X0) holds the PCI core, thus the board will be considered to hold two and a half PEs to implement the FIR filter, then there will be 792 * 2.5 = 1980 MFLOPS on each SLAAC1-V. In a best-case situation the ACS_API is considered to have negligible cost, then the SLAAC1-V Tower Of Power will reach 16 boards * 1980 MFLOPS (31680 MFLOPS) with this implementation.

The SLAAC1-V offers a 72-bit bus interface to the PEs, this bus can be used to input the Original and Accumulated data at the same time, then there will be no need to multiplex these two paths into the same bus any more; thus, area consumption on the PEs can be lowered due to removing the logic that took care of multiplexing.

Chapter 7: Conclusions

The goal of this thesis is stated as developing a fast, scalable, and reconfigurable FIR filter implementation. Constant multipliers were created to increase the speed of operation and reduce the area used on the chip. The Fast FIFO was created to increase the speed of data communication between the board and the host program. These two implementation optimizations helped in reaching the 26.7579 MFLOPS that was recorded in running the optimized direct-form FIR filter. Reconfiguration of the filter response was implemented in two ways, either by sending the filter coefficients through the data path or through reconfiguration of the multipliers, when using the constant multipliers. The design is scalable in the sense that it can be implemented on multiple boards while using the ACS_API to connect the boards together.

The Fast FIFO increased the data rate between the Host Program and the board with a factor of three. The Fast FIFO was also able to reduce the processor time needed during a transfer, the regular FIFOs required 95.28% of the processors time while the Fast FIFO only needed 39.54% of that time. The constant multipliers operated at a two fold higher speed than their JHDL counterparts, and required less area on the chip.

One of the most primary limitations of the design is area consumption. The maximum number of taps that can be placed on a PE is two taps. The full implementation of the FIR filter was tested on the WILDFORCE board. The Virtex chips on the SLAAC1-V board can accommodate larger designs than the X4000 Xilinx chips on the WILDFORCE board. Implementing the FIR filters on the SLAAC1-V can give better results in terms of speed and number of computations.

In the operation of the Fast FIFO, PE0 sends the size of the block to PE4, the word that carries the size has to pass through the logic of PE1, PE2, and PE3. This requires that the designs on these three PEs recognize that this control word should be considered invalid. A better implementation would not require these PEs to worry about such control words.

The constant multipliers use the modgen adders when there is a need for pipelining. The modgen adders can register their output without losing area on the chip. These adders do not operate for number representations larger than 32 bits and they are not mapped to work on the Virtex chips. Through the use of placement tools the regular adders can have a register at their output without extra area cost, implementing the constant multipliers with these tools will be of great benefit for the use of these multipliers on the Virtex chips.

Bibliography

[1] K. Ramachandran, *Unstructured Finite Element Computations on Configurable Computers*. Masters Thesis, Virginia Polytechnic Institute and State University, 1998.

[2] A. L. Walters, *A Scalable FIR Filter Implementation Using 32-bit Floating-Point Complex Arithmetic on a FPGA Based Custom Computing Platform*. Masters Thesis, Virginia Polytechnic Institute and State University, 1998.

[3] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing*. Prentice Hall 1996.

[4] Annapolis Micro Systems, Inc., WILDFORCE Reference Manual, 1997, 1998.

[5] Information Sciences Institute – East, SLAAC Project Page, http://www.east.isi.edu/projects/SLAAC

[6] B. Parhami, *Computer Arithmetic: Algorithms And Hardware Designs*. Oxford University Press, 2000.

[7] N. K. Bose, *Digital Filters: Theory and Applications*. Elsevier Science Publishing, 1985.

[8] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Prentice Hall, 1975.

[9] Brigham Young University's JHDL WWW Site, http://www.jhdl.org

[10] I.S. Board, "*IEEE standard for binary floating-point arithmetic*," Tech. Rep. ANSI/IEEE Std 754-1985, The Institute of Electrical and Electronics Engineering, New York, 1985.

[11] N. Shirazi, A. Walters, and P. Athanas, “*Quantative Analysis of Floating Point Arithmetic on FPGA Custom Computing Machines*,” IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, April 1995.

[12] Myricom High-Speed Computers and Communications, http://www.Myri.com

[13] K. Yao, *Implementing an Application Programming Interface for Distributed Adaptive Computing Systems*. Masters thesis, Virginia Polytechnic Institute and State University, 2000

[14] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, *"Implementing an API for Distributed Adaptive Computing Systems,"* Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, April, 1999.

[15] Xilinx, The Home Page For Programmable Logic, http://www.Xilinx.com

[16] The MathWorks home, Matlab 5.3.1 http://www.mathworks.com\products\matlab

[17] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[18] Steve Guccione, Delon Levi, and Prasama Sundararajan, “*JBits: Java based interface for reconfigurable computing.*” Second Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD’99), The Johns Hopkins University, Laurel, Maryland, Sep 1999.

Vita

Zahi S. Nakad

Zahi Nakad was born in December, 1976 in a small town called Jdita in the Bekaa Valley, Lebanon. After graduating from his high school, he was accepted to the Computer and Communication Engineering program in the American University of Beirut. After graduating with distinction in 1998, Zahi joined the Electrical Engineering program in Virginia Tech as a graduate computer engineering student. In the spring of 1999, Zahi joined the Configurable Computing Lab under Dr Mark Jones. He received his Master’s degree in 2000 and is now pursuing a PhD degree in Electrical Engineering at Virginia Tech.

Implementing an API for Distributed Adaptive Computing Systems

Mark Jones, Luke Scharf, Jonathan Scott, Chris Twaddle, Matthew Yaconis, Kuan Yao, Peter Athanas
Virginia Tech
and
Brian Schott
USC/ISI-East

1. Introduction

Many applications require the use of multiple, loosely-coupled adaptive computing boards as part of a larger computing system. Two such application classes are embedded systems in which multiple boards are required to physically interface to different sensors/actuators and applications whose computational demands require multiple boards. In addition to the adaptive computing boards, the computing systems for these application classes typically include general-purpose microprocessors and high-speed networks.

The development environment for applications on these large computing systems is not unified. Typically, a developer uses VHDL simulation and synthesis tools to program the FPGAs on the adaptive computing boards. External control for the board, such as downloading new configurations or setting clock speeds, is provided through a vendor-specific API. This API is typically accessed in a C host program that the developer must write in a high-level language environment. Finally, the developer is responsible for writing the networking code that allows interaction between the separate adaptive computing boards and general-purpose microprocessors. No tools are available for either debugging or performance monitoring in this agglomerated system. Development on these systems is time-consuming and platform-specific.

A standard ACS API is proposed to provide a developer with a single API for the control of a distributed system of adaptive computing boards, including the interconnection network. The API:

- is freely available,
- is intended to address a wide range of application requirements including embedded systems and cluster-based adaptive computing systems,
- provides a C-language binding,
- allows source code porting and scaling from small research platforms to large field-deployable platforms,
- provides a common API for a range of ACS platforms to promote portability,
- allows a developer to write a single host program that controls multiple boards,
- configures the network for communication between multiple boards,
- allows for direct reads/writes to each board's local memory,
- provides for multiple FIFO queues to be configured between boards,and
- is currently targetted to the SLAAC-1, SLAAC-2, and Wildforce platforms [6].

The ACS API is not a programming language for FPGAs; FPGAs are still configured using bitfiles generated by other development environments.

This paper presents the pertinent aspects of the API and the design philosophy, along with implementation details. Adaptive computing architectures and applications are discussed in Section 2. The structure of the API as well as short example host programs using the API are given in Section 3. The object-oriented implementation of the API is described in Section 4. Future work, including high-performance networking, is outlined in Section 5. Finally, the status of the implementation and how to access it are given in Section 6.

2. Distributed Adaptive Computing Architectures and Applications

The cluster computing paradigm is a cost-effective method of constructing small parallel computers using commercial off-the-shelf technology to exploit coarse- and medium-grain parallelism [2]. Adaptive computing systems are a COTS-based approach that exploits fine-grain and, to some extent, coarse-grain parallelism. The Tower of Power,

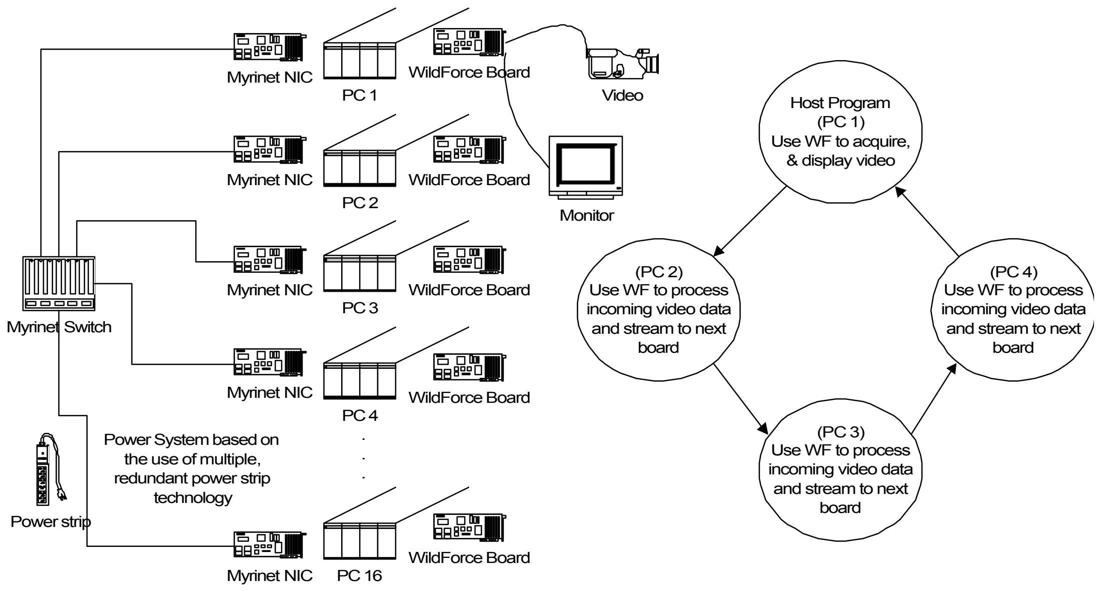

Myrinet NIC
PC 1
WildForce Board
Video
Host Program
(PC 1)
Use WF to acquire,
& display video
Myrinet NIC
WildForce Board
PC 2
Monitor
Myrinet Switch
Myrinet NIC
WildForce Board
PC 3
(PC 2)
Use WF to process incoming video data and stream to next board
(PC 4)
Use WF to process incoming video data and stream to next board
Myrinet NIC
WildForce Board
PC 4
(PC 3)
Use WF to process incoming video data and stream to next board
Power System based on the use of multiple, redundant power strip technology
Power strip
Myrinet NIC
PC 16
WildForce Board

Figure 1: The Tower of Power and a "ring" based image processing pipeline

is an example of one such cluster. As shown in Figure 1, the Tower of Power has sixteen Pentium II PCs each equipped with a WildForce board tightly coupled to a Myricom LAN/SAN card; the PCs are connected through a sixteen port Myrinet switch. A total of 80 XC4062XL FPGAs and memory banks are distributed throughout the platform, and are available as computing resources.

Such clusters, in various sizes, are readily available within the research community and suitable for many applications. The physical size of such a cluster, however, is not acceptable for use in embedded systems. A typical embedded system, such as the one shown in Figure 2, may have a single host microprocessor connected to a rack of densely populated ACS boards interconnected by a Myrinet switch. These large, embedded systems present at least two major difficulties. They present the same programming difficulties as the cluster model, but with the added complexity of a single microprocessor to control all of the nodes and provide all of the debugging information. Second, full-size embedded systems are quite expensive and beyond the means of most research laboratories; porting research codes from affordable laboratory clusters to full-size embedded systems is very difficult and time-consuming.

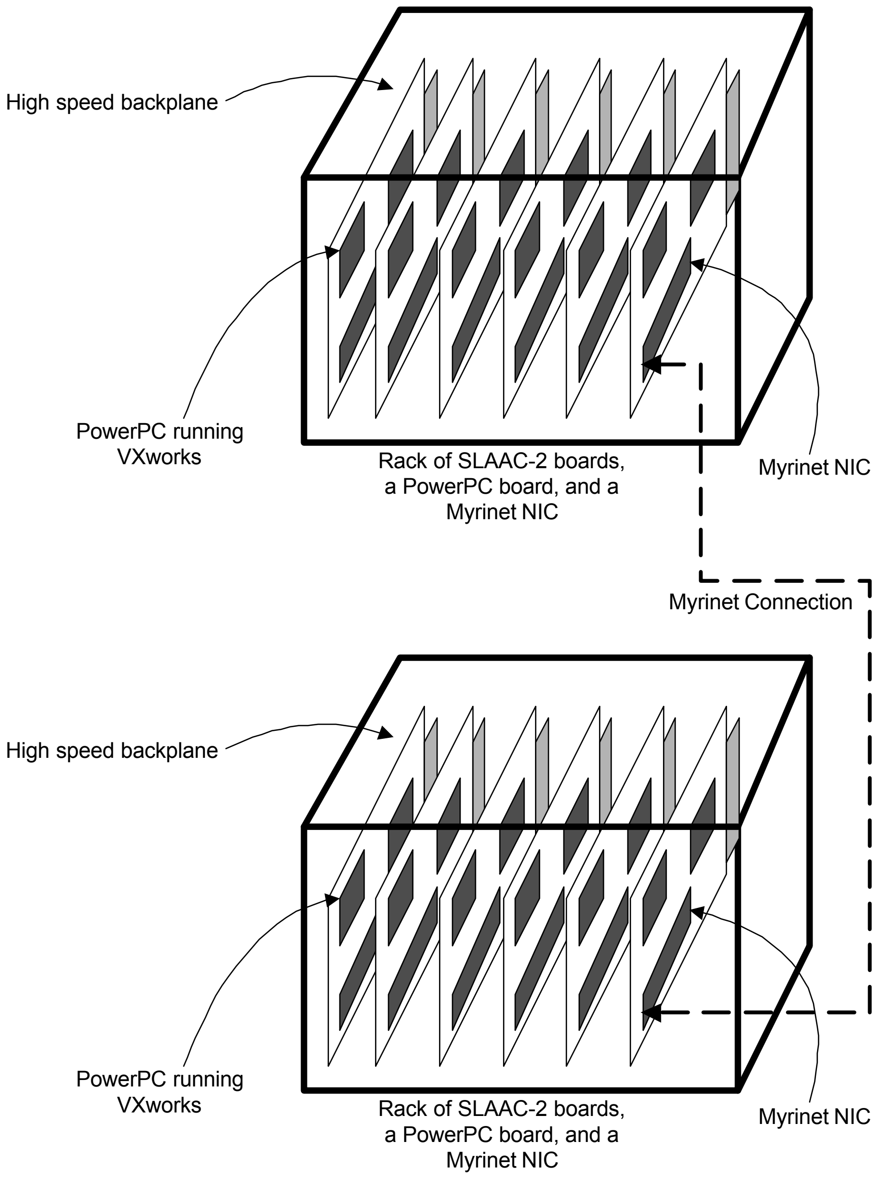


Figure 2: An embedded system with multiple SLAAC-2 boards

The ACS API provides an integrated development environment for exploiting clusters and embedded systems. It is the intent that applications developed on clusters using the ACS API can be ported without source code modification to larger, embedded systems. The API provides the structure and communication for coarse-grain parallelism while controlling the adaptive computing boards that provide the fine-grain parallelism. The primary model of coarse-grain parallelism supported by the ACS API is a channel-based model. A channel is a single logical FIFO queue connecting two computational nodes.

A user can construct an arbitrary system of channels and nodes. Thus, by using the API, the TOP, for example, can mimic large systolic arrays such as could be provided using multiple Splash-2 boards. The user is freed from the necessity of writing control programs on every computer to pass data manually; the API implementation controls the network and remote boards automatically after the channels are allocated. Furthermore, the API offers a consistent interface for both local and remote resources while preserving system performance.

The minimum system requirements assumed by the SLAAC API are a host CPU running a modern OS and an ACS board with logical or physical FIFOs. The current implementation also requires MPI-based network connectivity between all nodes of the system.

The API could be used to create an image processing application that requires more computational resources that are available on a single board. For example, one PC on an ACS cluster may be equipped with video in and out ports. To harness multiple boards in a deep image processing pipeline, a ring structure could be constructed in which video data is acquired in PC 1, processed in PCs 2-4, and then returned to PC 1 for output to a monitor. The communication in this structure is handled without intervention from the user after the channels are allocated.

3. API Description

The API is accessed from a C program called the *host* program. The host program provides for control of the entire system; the programmer need only write one host program no matter how many boards are in the system. The host program can access several classes of API calls allowing functionality such as system management, memory access, streaming data, and board management. Additional functions to allow for concurrent operations on multiple boards are also part of the API. One of the design goals of the ACS API is provide a simple API for the control of a complex system. A user is required to master a small set of commands to create a working system. This simplicity is providing by providing a set of reasonable default behaviors. An advanced user can gain more control of the system by altering these default behaviors through the use of a wider range of API functions.

3.1 System Management

The central component of the API is the specification and creation of a *system* object by the programmer. A system object is composed of *nodes* and *channels*. A node is a computational device, for example, an adaptive computing board. A channel is a logical FIFO queue between two nodes.

The first task in a host program is the creation of the system object. The code fragment in Figure 3 will construct the logical system in Figure 1. The program first creates two ACS data structures that describe the desired system object, in this case, a ring of four AMS WildForce boards. Note that these boards could easily be SLAAC-2 or Pamette boards; the API is not specific to any particular architecture. After calling the API initialization routine, the program makes a single call to *ACS_System_Create()* to create the system. Following an arbitrary user program that may contain more API calls, routines are called to destroy the ring system object and shutdown the API.

```
ACS_NODE nodes[4];   /* user structure to describe nodes  */
ACS_CHANNEL channels[4];  /* user structure for channels */
ACS_STATUS status;          /* status of ACS API commands */
ACS_SYSTEM ring;

for (int i=0;i<4;i++) { /* build a ring of 4 WildForce boards */
      nodes[i].model = “WF4”;
      channels[i].src_node = i;
      channels[i].src_port = 0;
      channels[i].dest_node = (i+1) % 4;
      channels[i].dest_port = 1;
}
ACS_Initialize(argc, argv, &status);  /* must be first API call */
ACS_System_Create(&ring, nodes, 4, channels, 4);
/* user program that accesses “ring” object */
ACS_System_Destroy(ring);
ACS_Finalize();                     /* must be last API call */
```

Figure 3: Code fragment for creating a "ring" system object.

In addition to the static system creation illustrated in Figure 3, the API also has features for altering a system at runtime. Node and channels may be added or deleted after the creation of a system object through API calls. Note that multiple host processes are also possible in the API, but are not discussed in this paper for the sake of clarity.

3.2 **Board Management**

Once the system object has been created, the boards can be configured and controlled via the API. The code fragment in Figure 4 sends a bitstream to each board as specified in a configuration data structure. This configuration data structure includes information on which processing elements to configure as well as board-level configuration information such as crossbar switch settings. After configuration, the code fragment sets the clock speed, starts the clock, and then sends a reset signal. Finally, as described further in Section 3.3, the API provides calls for writing directly to the memory of a board. The second loop illustrates this call as well as the capability of sending interrupt signals of various types to each board.

Also included for board management are routines to query the board, including functions

```
for (int i=0;i<4;i++) {
      /* send bitstream for each ACS board */
      ACS_Configure(config[i],i,ring,&status);
      ACS_Clock_Set(clock,i,ring,&status);/* set clock speed    */
      ACS_Run(i,ring,&status);            /* start clock        */
      ACS_Reset(i,ring,&status);          /* send reset signal */
}
for (int i=0;i<4;i++) {
      /* write initial data to each board's memory */
      ACS_Write(databuf[i],datalen[i],i,brd_addr[i],ring,&status);
      /* then send an interrupt (or inta) signal #1 to the board */
      ACS_Interrupt(i,1,ring,&status);
}
```

for readback and querying the clock speed. Current plans include building upon these routines to provide significant debugging capability within the API.

Figure 4: Code fragment for configuring and writing to ACS boards

3.3 Memory Access

The API contains routines for read and write access to the memories of all boards, local and remote. The use of the *ACS_Write()* function is illustrated in Figure 4. Also included, to reduce network traffic, is a memory copy command that causes a block of memory to be copied from one node to another node rather than using a read followed by a write. These commands allow data to be sent outside of the channel-based system model; they put the burden of explicitly specifying all data movement solely on the developer. Nevertheless, they can be quite useful for sending initialization data or retrieving accumulated data directly from boards, operations for which the channel model is not naturally suited.

3.4 Streaming Data

The channel-based communication model requires the user to explicitly control only the initial entry and final exit of data from the system. Two primary commands, *ACS_Enqueue()* and *ACS_Dequeue()*, are required to control communication. The use of these commands is illustrated in Figure 5 where they control the data flow in the ring system that was created by the code fragment in Figure 3. The user can specify the behavior of each of the channels with additional API function calls, but is not required to do so. Such behavior can include the buffer size associated with a channel as well as the underlying communication mechanism.

```
/* use the ring to process the required number of images */
for (int i=0;i<NUM_IMAGES;i++) {
      /* send image onto channel associated with port 0*/
      ACS_Enqueue(image[i],IMAGESIZE,0,ring,&status);
      /* get resulting image from channel associated with port 1 */
      ACS_Dequeue(result_image[i],RESULT_SIZE,1,ring,&status);
}
```

Figure 5: Code fragment demonstrating channel-based communication

3.5 Non-blocking Commands

An advanced feature of the API is a mechanism for issuing a set of non-blocking commands. Up to this point, the API functions discussed in the paper have all been blocking. For example, the *ACS_Write()* commands in Figure 4 occur one after the other with the host program blocked during the execution of each write. Through the *ACS_Request()* function, a user can specify a sequence of API functions to be executed as a set; this sequence is called a request and may include commands to read/write/copy memory, raise a reset line, or send an interrupt (the set of possible commands will be expanded). One a request has been created, it can be committed to execution using *ACS_Commit(). ACS_Commit()* issues the commands, creates a handle, and returns control to the user. While those commands are executing, the user may perform other operations. Completion of the set of commands can be checked using *ACS_Test()*, or can be waited upon in a blocking fashion using *ACS_Wait()*. Once created, a request may be committed to execution multiple times. Benefits of the request mechanism include improved efficiency by overlapping user task execution with API task execution, combining multiple commands to reduce network overhead, and re-using command sequences to reduce API overhead.

3.6 Group Operations

The API also allows for certain commands to result in broadcasts of data rather than simple point-to-point transfers. By specifying *ACS_ALL*, rather than an individual node number, the *ACS_Configure()* command can become a broadcast to all nodes, allowing for a single command to configure all the ACS boards in the system. This broadcast function would be quite useful on a homogeneous system, but would not be appropriate on a system with both SLAAC-1 and SLAAC-2 ACS boards. The group management functions in the API can be used to specify groups of nodes in the system. Group identifiers can be used to transform broadcasts into multicasts. For example, the SLAAC-1 boards could be programmed in one *ACS_Configure()* call and the SLAAC-2 boards in another. These group operations do not affect channel-based communications.

3.7 The Interface on the Processing Elements

To this point, the focus of this paper has been on the host-side programming interface. The interface from the perspective of the processing elements is diverse, but not unlimited. Some obvious prerequisite capabilities on the part of the processing elements are the ability to observe reset lines and manipulate/observe interrupt lines. If memory is present, the processing elements can communicate with the host program by reading/writing that memory. Most importantly, the processing elements on a board (or a subset) must be able to read and write to a set of numbered (perhaps logical) FIFOs to support the channel-based communication model. Unlike the other capabilities, this feature may not be available on all systems and, if present, will not provide an unlimited number of FIFOs. An aspect of porting the API to any new board architecture is the provision of a FIFO mechanism; such a mechanism can, for example, be provided by a combination of memory reads/writes and interrupts. It is the responsibility of the API implementation itself to manage a limited number of physical FIFOs.

4. API Implementation

Control of the system across multiple computers is accomplished by using a single process on every computer. The host program serves this purpose on the computer on which it runs. Other computers in the system run a *control* process. A control process is responsible for executing commands initiated by API calls, monitoring the local adaptive computing board, and communicating with other control processes. Each of the processes is multi-threaded to allow for concurrent communication and computation; such multi-threading also allows the host process to execute the control process functions on the computer where the host process is running.

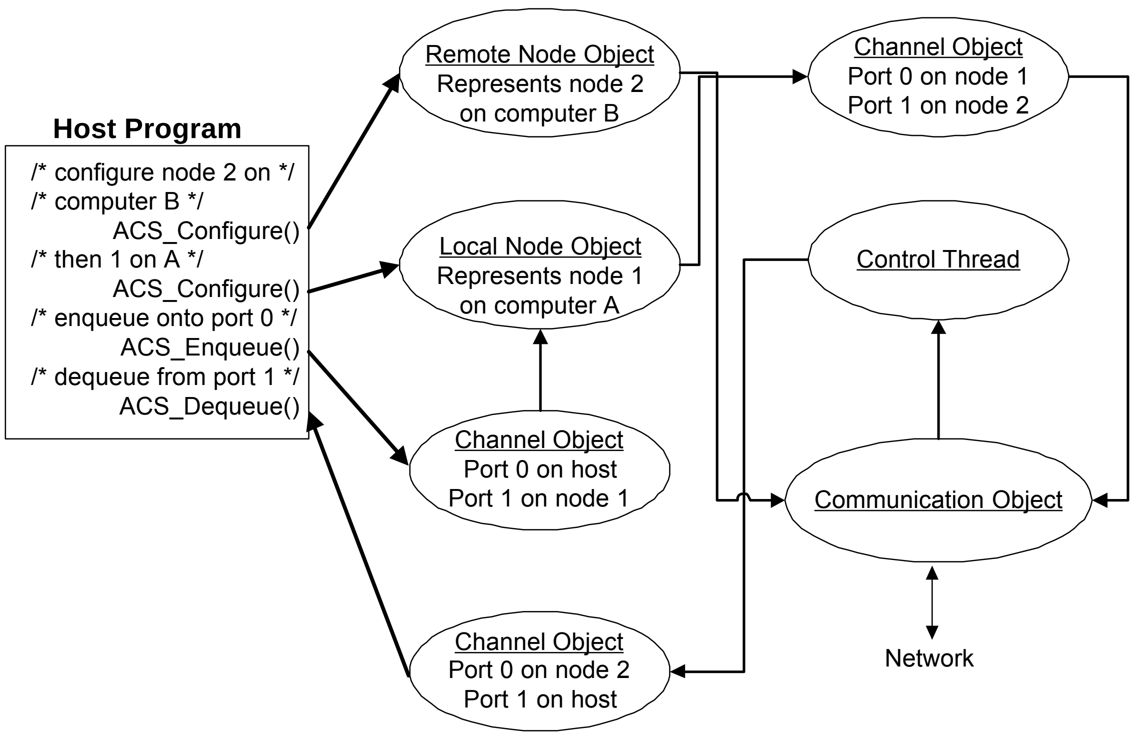
To be of use in the adaptive computing community where the technology is changing rapidly and experimentation is of prime importance, any implementation of this API should be easily extensible. Such extensibility is accomplished by using an object-oriented approach to implementation, as is possible with C++. The identification of objects arises naturally from the specification of the API. As outlined earlier, the *system* object is a collection of *node* and *channel* objects. A channel object represents a FIFO queue connecting one node to another node and can contain a buffer as well as settings specific to control flow on this FIFO. A node object represents a computational device in the system.

Two objects not directly viewed or manipulated by the user are the *communication* object and the *world* object. The communication object accomplishes all communication between processes on different computers. Different communication objects can be used to allow functionality in a heterogeneous network. The only communication object in the current implementation uses MPI for communication. MPI is itself a standard API for communication between parallel processes [5]. It was chosen as the first communication object in this implementation because of its ubiquitous nature and the availability of high-performance implementations [3]. The world object is used to encapsulate and maintain information about the computing environment in which the API is running. For example, the world object will contain a list of all the control processes and host processes running as well as how to communicate with those processes. Further, it will contain a list of all the adaptive computing boards managed by each control process. Future extensions to the API include a collection of routines to query the world object so that the user can dynamically create system objects based on which types of boards are available.

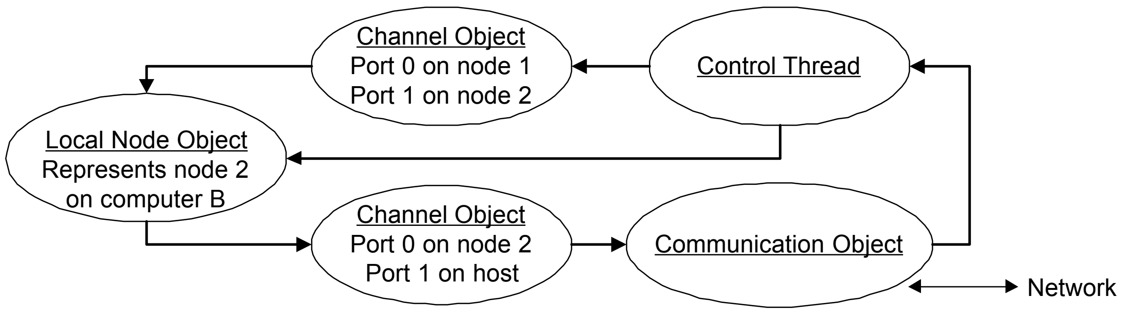
The core of the API implementation is written as operations on these objects. The classes associated with these objects, including virtual function definitions, are defined as part of the core implementation. By taking advantage of inheritance and encapsulation, the distinction between local and remote boards is easily hidden, and new types of boards and communication systems can be seamlessly included. For example, to extend the API to allow control of a new board, a developer just creates a class that inherits the node object and implements all of the virtual functions to allow for control of the new board; the rest of the API implementation remains unchanged.

The need for high performance from the API cannot be lost in the quest for an easily extensible implementation. Of particular importance is the need to avoid introducing unnecessary overhead into the implementation. A potential pitfall in any interprocessor communication system is the introduction of multiple copies of large buffers. The specific method for avoiding buffer copies, particularly ones implicit in calls to an underlying communication system, are specific to the type of communication object used. The API implementation, outside of the communication object, will not introduce extra copies of large buffers. Further savings in overhead can be accomplished by recognizing when commands or data are being sent to a local board as opposed to a board on a remote computer. Fortunately, the object-oriented implementation can accomplish this by simply providing a *remote* node object and a *local* node object that each inherit from the node object. Actions by the API are performed on a node object without regard to local/remote considerations, but the correct node functions are called automatically depending on the whether the node is local or remote. This results in a logically simple implementation that introduces no unnecessary overhead for local operations.

An illustration of the objects in the implementation and their interaction in a typical API operation is given in Figure 6. The objects in this figure represent a host program and node 1 executing on computer A and node 2 executing on Computer B; these objects are connected by three channels to form a ring communication structure. The example host program illustrates the basic communications that occur when the nodes are configured as well as the communications that occur when communications are initiated in the ring.



Objects and threads on Computer A which runs the host program and contains node 1. The host program, node 1, and node 2 are connected in a ring.



Objects and threads on Computer B which contains node 2.

Figure 6: Description of objects and threads on two computers.

5. Future Work

Proposed future extensions to the API and its implementation include high-speed networking, support for RTR, support for debugging and performance monitoring, system level management of multiple programs, and integration with systems such as JHDL [1].

The current implementation of the API uses MPI [5] for interprocessor communication; the available high-performance versions of MPI are suitable for most applications [3]. However, some applications require a tighter coupling of the communication board to the computational elements. Some embedded systems designs have communications processors on the same boards as the computational elements. While the API and its implementation are designed such that the host processes will likely require the support of a traditional OS, the control processes do not require the full services of a traditional OS. The next step of the API implementation is to carry out the functions of a control process on two PCI cards, a Myrinet interface card and an ACS card, without the intervention of the CPU or OS.

Also of significant importance is the support for efficient run-time reconfiguration (RTR) of adaptive computing devices. As currently specified, the API requires that a host process initiate any device reconfiguration via an *ACS_Configure()* function call. Further, the current implementation requires that every *ACS_Configure()* function call result in the configuration being sent by the host process to the device to be reconfigured. Future versions will address two modes of efficient RTR. The first mode is host-initiated RTR; in this mode, the host process is always the initiator of reconfiguration. Efficient support for this mode will be provided by allowing control processes to cache several configurations. When *ACS_Configure()* is called, the cache will be checked for the desired configuration and if the configuration is present, then the network transfer can be avoided. The second mode, data-driven RTR, is more complex to support. In this mode, reconfiguration is driven by the data that is encountered. For example, different filters may be downloaded depending on the light-levels encountered in an image. The host process cannot drive such reconfiguration. In this case, the programmer must be able to describe more complex mechanisms to the control processes, perhaps finite state machines, to manage the reconfiguration. Note that such a mechanism will be required for single board/computer systems as well as the multi-board/computer systems addressed in this paper.

As noted in the integration, no tools exist for performance monitoring and debugging of applications on distributed adaptive computing systems. Board-level tools, such as BoardScope, can provide valuable support for debugging an application on a single board [4]. Extensions to the API to support debugging and performance monitoring can facilitate the task of creating integrated system-level tools. For example, the current API already includes a function that allows for readback of any node on the system. Other types of support could include functions to query the status of the channels, channel throughput statistics, and usage statistics on processing elements.

Also of concern is the management of multiple concurrent programs. For example, if two processes try to configure the same adaptive computing board and no management software intervenes, then both processes may fail in unpredictable ways. The immediate solution to this problem is simply to lock one of the processes out of adding that node to its system object. A time-sharing feature, however, is likely to be more useful and more interesting. For example, on systems that do not support faster RTR, it would be useful to checkpoint jobs that have been running a long time in favor of new jobs. On systems that support fast RTR, it should be possible to context switch between jobs in a fashion similar to traditional operating systems. The combination of fast RTR and efficient system software would provide a useful virtual adaptive computing system.

The API can serve as the target of higher-level tools as well as a direct implementation language. One such higher-level tool, JHDL, provides a portable, integrated environment for programming a single adaptive computing board as well as the interaction of that board with a host system [1]. A developer can use JHDL to write a host program, program the FPGAs, and debug the combined application. JHDL does not, however, currently provide support for system level applications nor does it provide synthesis capabilities. Because of its use of an existing programming environment, Java, JHDL could be expanded to included support for the large, distributed systems targeted by the ACS API. Such support can be more easily provided if JHDL can use the ACS API as a target. The use of the ACS API as a target, as well as its extension, can be facilitated by exposing the objects and functions of the underlying implementation in a formal way to developers.

6. Conclusions

The API specification, source code, and supporting documents can be accessed on the Virginia Tech Configurable Computing Lab WWW site at http://www.ee.vt.edu/~ccm. The first version of the API implementation is complete. The first version uses MPI for interprocessor communication. Near-term plans include porting the ACS API implementation to Linux and adding node objects for the SLAAC-1 and SLAAC-2 ACS boards.

The ACS API and the implementation have applications outside of the control of adaptive computing boards. In particular, they can be used to a control a heterogeneous collection of devices in an embedded system. Instead of configuring an adaptive computing device, *ACS_Configure()* could download a configuration to microcontroller or a FFT board. Because of the object-oriented nature of the implementation, this functionality can be accomplished via the addition of another node object class. To demonstrate this functionality and for use in rapid prototyping, the current implementation has a *dummy* node object class that defines all of the virtual functions for a microprocessor. A developer can insert arbitrary C code in this dummy node so that it can emulate the behavior of any device; a system-level application can be quickly constructed and dummy nodes gradually replaced to increase performance without changing the functional behavior of the system.

7. References

1. P. Bellows and B. Hutchings, “JHDL-An HDL for Reconfigurable Systems,” *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
2. D. S. Katz, T. Cwik, B. H. Kwan, J. Z. Lou, P. L. Springer, T. L. Sterling, and P. Wang, "An Assessment of a Beowulf System for a Wide Class of Analysis and Design Software," *Advances in Engineering Software*, Vol. 29(3-6), pp. 451-461, 1998.
3. M. Lauria and A. Chien, “MPI-FM: High Performance MPI on Workstation Clusters,” *Journal of Parallel and Distributed Computing*, Vol. 40(1), pp. 4-18, 1997.

4. D. Levi and S. A. Guccione, “BoardScope: A Debug Tool for Reconfigurable Systems,” In John Schewel, ed., *Configurable Computing Technology and its Use in High Performance Computing, DSP, and Systems Engineering, Proc. SPIE Photonics East*, Bellingham WA, pp. 239-346, 1998.
5. Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard", *International Journal of Supercomputing Applications*, Vol. 8(3/4), 1994.
6. B. Schott, C. Chen, S. Crago, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Architectures for System-Level Applications of Adaptive Computing," submitted to the *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999.

Implementing an Application Programming Interface for Distributed Adaptive Computing Systems

by

Kuan Yao

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

May 31, 2000
Blacksburg, Virginia

Keywords: MPI, Adaptive Computing, FPGA, API

Implementing an Application Programming Interface for Distributed Adaptive Computing Systems

Kuan Yao

Committee Chairman: Dr. Mark T. Jones

Committee Members: Dr. Peter M. Athanas, Dr. Scott F. Midkiff

The Bradley Department of Electrical and Computer Engineering

(ABSTRACT)

Developing applications for distributed adaptive computing systems (ACS) requires developers to have knowledge of both parallel computing and configurable computing. Furthermore, portability and scalability are required for developers to use innovative ACS research directly in deployed systems. This thesis presents an Application Programming Interface (API) implementation developed in a scalable parallel ACS system. The API gives the developer the ability to easily control both single board and multi-board systems in a network cluster environment. The API implementation is highly portable and scalable, allowing ACS researchers to easily move from a research system to a deployed system. The thesis details the design and implementation of the API, as well as analyzes its performance.

ACKNOWLEDGMENTS

I am indebted to Dr. Mark Jones, my major professor and thesis chair, for his inspiration and guidance at various stages throughout this project. Dr. Jones has generously given much of his time to talk with me and to read and consider the drafts I have prepared. His expertise and advice are invaluable in this project.

I would also like to acknowledge and thank the members of my committee Dr. Athanas and Dr. Midkiff for reading my thesis and providing insightful opinions. For help with the proofreading, I am indebted to Katherine for her helpful suggestions and enormous patience.

A special thanks is due to my parents, my uncle, and my wife, who have always held great faith in me and extended generous support to my study and my life.

Table Of Contents

List of Figures

List Of Tables

1. Introduction

1.1 Background

Adaptive Computing Systems (ACS) dynamically reconfigure their logic and/or data paths to match application requirements. Instead of using traditional microprocessors, they use reconfigurable computing boards, which can provide significant improvements in speed over general-purpose microprocessors for many applications. Because the modern cluster computing community uses workstations and COTS high-speed networks to build high-performance parallel systems [1], a logical way to build scalable parallel ACS systems is to cluster FPGA-accelerated workstations.

The Tower of Power (ToP) at Virginia Tech is a typical example of a scalable parallel adaptive computing system [2]. The ToP has sixteen Pentium II PCs. Each PC is equipped with a WildForce board [3] and tightly coupled to a Myricom LAN/SAN card [4]. These PCs are connected through a sixteen-port Myrinet switch. A total of 80 Xilinx XC4062XL FPGAs [5] and memory banks are distributed throughout the platform. Applications for the ToP include image processing, large-scale histogram matching, and large matrix multiplication. Figure 1.1 shows the ToP architecture and a pipeline programming model.

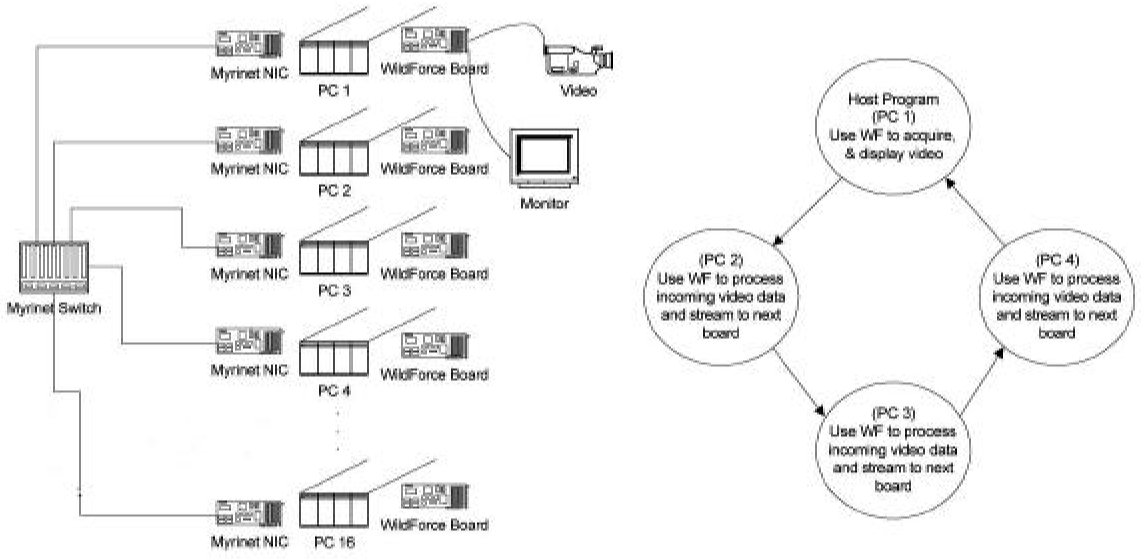

Myrinet NIC
PC 1
WildForce Board
Video
Myrinet NIC
PC 2
WildForce Board
Monitor
Myrinet NIC
PC 3
WildForce Board
Myrinet Switch
Myrinet NIC
PC 4
WildForce Board
Myrinet NIC
PC 16
WildForce Board
Host Program (PC 1) Use WF to acquire, & display video
(PC 2) Use WF to process incoming video data and stream to next board
(PC 4) Use WF to process incoming video data and stream to next board
(PC 3) Use WF to process incoming video data and stream to next board

Figure 1.1: Virginia Tech ToP architecture and a "ring" based image-processing pipeline [11]

1.2 Problem

Developing large applications on such a distributed adaptive computing system environment typically requires several steps. First, the developer needs to use VHDL [6] or some other hardware description language to specify the design. This hardware description language is then synthesized to create the FPGA configuration. In addition, the user must write a high-level language program to control the board for performing tasks such as downloading configuration files to the board, setting up clock rates, and sending data to the board [11].

Finally, the developer has to write network communication code to support the parallel system. This last step poses a major problem because building applications in this way is time-consuming and platform-specific. Moreover, no tool exists to debug and monitor a distributed ACS system. An additional difficulty typically encountered is that developers are generally not familiar with both HDLs and the development of high-performance parallel systems.

1.3 Statement

To address the problems posed in Section 1.2, a scalable Application Programming Interface (API) and a runtime software system was designed and developed to support applications of the network-distributed ACS system. This common API gives the developer the ability to control single board, multi-board, and heterogeneous board systems through the same interface, thus making the application for the ACS system scalable and portable. This thesis will describe in detail the API, its implementation, and the performance of that implementation.

1.4 Thesis organization

The thesis is organized in the following manner. Chapter 2 surveys related research and background material. Chapter 3 gives a description of the API along with a rationale for the design. Chapter 4 provides a detailed description of the implementation of the API. Chapter 5 describes the testing procedures and performance results. Chapter 6 summarizes the work and gives directions for future research.

2. Introduction to Adaptive Computing and MPI

2.1 Overview

A parallel ACS system, such as the Tower of Power, uses multiple adaptive computing boards as its processing elements instead of traditional CPUs. The adaptive computing devices are the key components of an ACS system. This chapter gives an introduction to some of the adaptive computing platforms used in this research project, as well as information on related research. The Message Passing Interface (MPI) [9], a standard message passing parallel programming tool used in this project, is also introduced in this chapter.

2.2 Adaptive Computing Devices

2.2.1 WildForce

The Wildforce board is a reconfigurable board developed by Annapolis Micro Systems Inc. It is used in the Tower of Power (ToP) as attached reconfigurable computing boards. The Wildforce board has five FPGAs, one control-processing element (CPE), and four array-processing elements (PE). A mezzanine connector connects all the memory elements in each PE. Both the PE and host CPU can access these memory elements through a Dual Port Memory Controller (DPMC). All PE' s in the board are connected by a crossbar interconnect network. The crossbar can be configured by the control PE to allow the broadcasting or transferring of data between specified PEs. Each PE can also communicate with its neighboring PE through the systolic bus. The control PE, PE1, and PE4 provide FIFO buffers that can exchange data with the host CPU. They also can communicate with an external I/O. Figure 2.1 below shows the architecture of the Wildforce board [10].

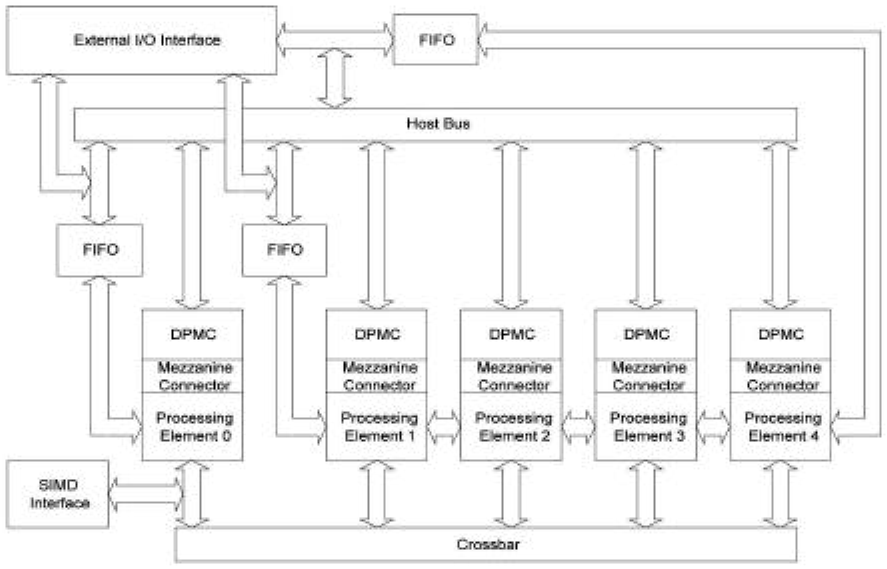


Figure 2.1: Wildforce board architecture [10]

2.2.2 Splash 2

The Splash 2 [12] project began at the Supercomputing Research Center in September 1991 and ended in the spring of 1994. The Splash 2 is an attached processor system using Xilinx XC4010 FPGAs as its processing elements. Figure 2.2 illustrates the system architecture of Splash 2. The system consists of three parts: a SPARCstation 2 host computer with an Adaptive Board; a Splash 2 Interface board; and up to 13 Splash 2

Array Boards. The Adaptive Board extends the host computer' s address and data bus to allow the host program to directly access the memory in the Splash 2 system. Each Array Board contains 17 Xilinx XC4010 FPGA chips as its processing elements. Sixteen of them, X1 through X16, form the processing array and are connected by a crossbar. X0 provides the control functions to the crossbar and broadcasts data in SIMD bus to all the other FPGAs.

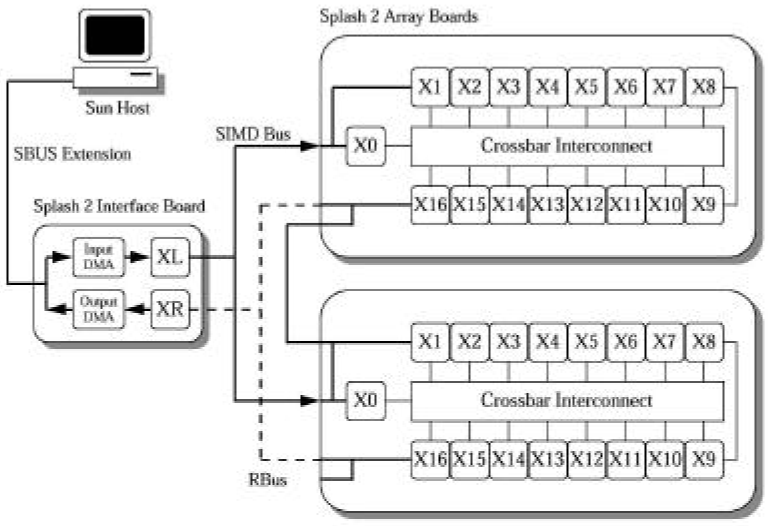


Figure 2.2: Two board Splash system [10]

The Splash 2 system was primarily designed to support applications that have SIMD or a broadcast-of-data model and a linear model. In the SIMD model,the 36-bit-wide data path from the Interface Board to each Array Board serves as a SIMD bus. The Xilinx chip X0 on each Array Board can then broadcast the SIMD Bus data to the other FPGAs on its Array Board. Pattern recognition is a typical application using this model [12].

In the linear model, the 36-bits wide data path can be used to transmit data from the Interface Board to the first FPGA on the first Array Board. The data can then be moved linearly through X1 to X16, then to the first FPGA on the second board, and so forth. Finally, when the data reaches the last FPGA on the last board, it goes back to the Interface Board. This computation model is good for applications that can be deeply pipelined, for example, image processing.

Every Splash 2 application may be divided into three main parts: the portions that run on the Array Boards; the Interface Board; and the host computer. To build a Splash 2 application, the developer first needs to use VHDL as a programming language to write configuration files for the Processing Elements, X1 through X16, the Control Element, X0, and the crossbar in each Array Board. The developer then needs to write configuration files for the Interface Board. Finally, he needs to provide a C program running on the host computer, which is responsible for downloading the configuration data to the FPGAs, establishing input and output data streams, and controlling the clock.

Developing an application for Splash 2 is like designing a program for a massive parallel computer. In the data parallel (SIMD) model, the programmer must control the data layout among the Array Boards, which is critical to the performance of parallel computing. In the pipelined model, the control layout is critical. Therefore, a good algorithm must be used to partition the tasks and data among the PEs so as to maximize the efficiency of the inter-PE communication. However, there are no automated tools that can do this task, the programmer must perform the partitioning manually.

2.2.3 SLAAC Project

The System Level Application of Adaptive Computing (SLAAC) project is sponsored by the Defense Advanced Research Projects Agency (DARPA) Adaptive Computing System program [8]. The goal of SLAAC is to define an open, distributed, scalable, adaptive computing system architecture based on a high-speed network cluster of heterogeneous, FPGA-accelerated nodes.

The SLAAC architecture is based upon a high-speed network of ACS-accelerated nodes. A high-speed network cluster of traditional desktop PCs, where each is accelerated with some form of an ACS accelerator (such as a PCI FPGA-board), is called a Research Reference Platform (RRP). The RRP is an inexpensive, readily available platform for ACS development that tracks advances in workstations, adaptive computing, and cluster computing [14]. The Tower of Power at Virginia Tech is a good example of an existing RRP.

Similar to Splash 2 and Wildforce, the SLAAC architecture is an attached processor system consisting of FPGA's and fast local memories. SLAAC-1 is an attached FPGA-based accelerator on a full-sized 64-bit PCI board. SLAAC-1 features one user-programmable Xilinx 4085 device, two user-programmable Xilinx 40150 devices, and ten 256Kx18 100MHz ZBT synchronous SRAM's. The SLAAC-1 architecture is composed of one single interface FPGA (labeled IF), and three user-programmable FPGA's (labeled 'X0', 'X1', and 'X2'). The IF chip serves as a stable bridge to the host system bus. It provides the configuration, clock, and control logic for the 'X0', 'X1' and 'X2'. The role of the attached system is to program the user FPGA's and control the system. The SLAAC-1 can execute either synchronously with the host or asynchronously with the DMA channels to transfer data to and from the host memory. Through the clock generator and FIFO's from the IF, the user FPGA's are allowed to operate from a single data-synchronous clock in either mode of operation [14].

SLAAC-2 is an attached FPGA-based accelerator on a 6U VME mezzanine board. SLAAC-2 features two user-programmable Xilinx 4085 devices, four user-programmable Xilinx 40150 devices, and twenty 256Kx18 100MHz ZBT synchronous SRAM's. SLAAC-2 is actually two "independent" SLAAC-1 bit-file compatible accelerators on a single board [14].

As part of the SLAAC project, the Configurable Computing Lab at Virginia Tech is building a common API which supports developing applications for the SLAAC system. The development of this API is discussed in detail in later chapters.

2.3 Message Passing Interface

Message passing is a programming paradigm used widely on parallel computers, especially on scalable parallel computes with distributed memory. Message passing is also used on networks of workstations. The Message Passing Interface (MPI) is used as a standard for writing message-passing programs. The goal of MPI is to develop a widely used standard for writing message-passing programs. As such, the interface should establish a practical, portable, efficient, and flexible standard for message passing.

With interfaces to programming languages such as Fortran and C, MPI has great portability. This means that the same message-passing source code can be executed on a variety of machines as long as the MPI library is available. MPI also has the ability to run transparently on heterogeneous systems, where the processors have distinct architectures. Through the ability to span such a heterogeneous system, MPI provides a virtual computing model that hides many architectural differences. The user need not worry whether the code is sending messages between processors of like or unlike architectures. The MPI implementation will automatically do any necessary data conversion and utilize the correct communication protocol.

MPI was designed to enable the overlap of communication and computation, thus hiding communication latencies. This is achieved by the use of non-blocking communication calls which execute communication in the background. MPI also supports scalability through several of its design features. For example, an application can create subgroups of processes that allow collective communication operations only on this subgroup, such as broadcast, multicast, gather, and scatter. MPI guarantees that the underlying transmission of messages is reliable. The user need not check if a message is received correctly [15].

The Tower of Power is a network cluster system. Each workstation in the system has its own memory and processing unit. Thus message passing is the only way to run applications on the system (versus shared memory). There are two major reasons that MPI was chosen for the underlying communication routines of the ACS API implementation. First, MPI provides the ability to control the distributed system at a high level. In the Splash 2, the programmer has to control multiple boards through configuration files for setting up the Control Element and the crossbar, in marked contrast to MPI. This high-level control ability makes an ACS API application easy to develop and highly scalable. Second, MPI vendors provide a variety of libraries to support different platforms. This makes the ACS application very portable to different environments, including embedded processing environments.

2.4 Summary

The Wildforce, SLAAC-1, and SLAAC-2 are all adaptive computing devices that can dynamically change their logic and data path to match the applications' requirements. They are used by the SLAAC system as attaching processing elements to create supercomputer-like performance in various applications. The ACS API is developed to support writing applications to run on such systems. MPI, as a standard for writing message-passing programs, is used in the ACS API for low-level communications because of its portability and ease-of-use.

3. ACS API Descriptions

3.1 Overview

The design goal of the ACS API is to provide the developer with a simple, system-level API to control a complex distributed system. The system-level of the API focuses on two important issues in the ACS application: scalability and portability. Scalability requires that the application source code can be easily ported and scaled from small research platforms to large field deployable platforms. Portability requires that the API should provide a common interface for a range of ACS platforms, including embedded system and cluster-based adaptive computing systems. The simplicity of the API is achieved by providing a set of reasonable default behaviors for controlling the system. Meanwhile, through a wider range of API functions, an advanced user can alter these default behaviors – thus attaining more control of the system. The API implementation is open-source, which gives the users the ability to easily add new nodes and new functionality to the API.

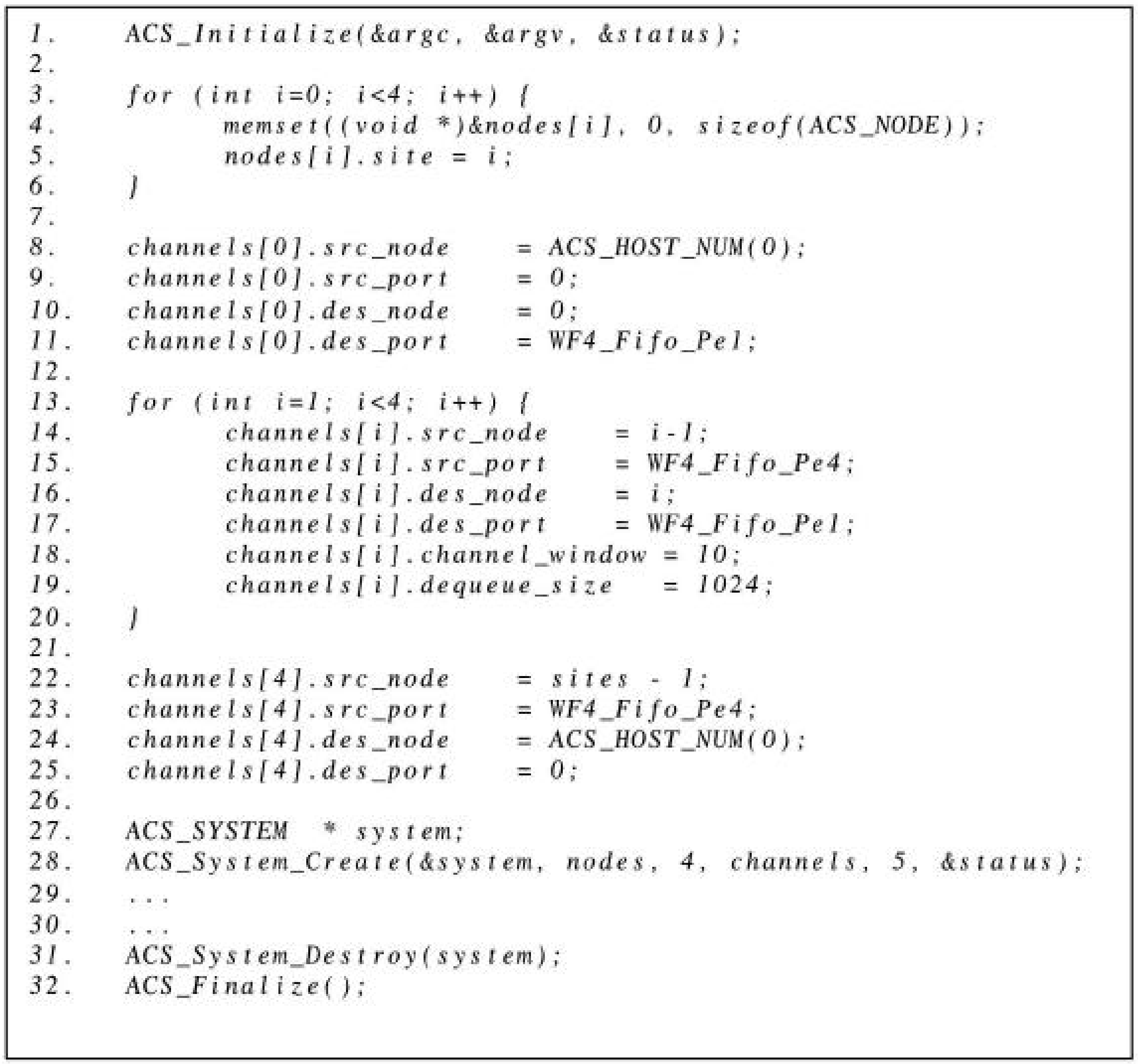
The programming model defined in this API is a single application program written in high level language, usually C, that allocates and controls a heterogeneous set of distributed adaptive computing devices. Based on the functionality, the APIs can be divided into five categories: system management, board management, data accessing, group functions, and non-blocking functions. The following sections give a short description of the ACS API, its structures, components, and show how a program controls the ACS system through the API. The full specification of the API can be viewed in Appendix A1. ACS API Specification.

3.2 System Management

The System Management APIs allow the user to initialize the ACS system. They include *ACS_Initialize(), ACS_Finalize(), ACS_System_Create(), and ACS_System_Destroy().* The primary component of the API is to let the programmer specify and create a *system* object, which consists of *node* objects and *channel* objects. A node is defined as a computational unit, for example, a reconfigurable board such as the Wildforce board. A channel is defined as a logical FIFO queue between two nodes. Most of the remaining API calls use this *system* object for controlling the ACS system.

The first step of a host program is to initialize the ACS system. This includes parsing command lines, invoking processes in other workstations, establishing network communication, and initializing global data structures. This step is accomplished by calling *ACS_Initialize()*. The program then makes a call to *ACS_System_Create()* to create the system object, along with the node and channel objects. The user needs to provide node and channel information, such as the location of the node in the network or the source and destination of a channel. The host program then follows an arbitrary user program with more API calls. Finally, the host program will call *ACS_System_Destroy()* to destroy the system object and *ACS_Finalize()* to shutdown the ACS system.

Figure 3.1 shows a fragment of an initialization procedure for an ACS program. The first line calls *ACS_Initialize()* to initialize the ACS system. Lines 3 to 5 are for setting up the node information and telling the system in which workstation the node resides. Lines 8 to 25 are for setting up the channel information. The channels link the node into a ring, as shown in Figure 1.1. Lines 31 and 32 are to close and terminate the ACS system.



```
1.     ACS_Initialize(&argc, &argv, &status);
2.
3.     for (int i=0; i<4; i++) {
4.             memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
5.             nodes[i].site = i;
6.     }
7.
8.     channels[0].src_node     = ACS_HOST_NUM(0);
9.     channels[0].src_port     = 0;
10.    channels[0].des_node     = 0;
11.    channels[0].des_port     = WF4_Fifo_Pe1;
12.
13.    for (int i=1; i<4; i++) {
14.            channels[i].src_node     = i-1;
15.            channels[i].src_port     = WF4_Fifo_Pe4;
16.            channels[i].des_node     = i;
17.            channels[i].des_port     = WF4_Fifo_Pe1;
18.            channels[i].channel_window = 10;
19.            channels[i].dequeue_size   = 1024;
20.    }
21.
22.    channels[4].src_node     = sites - 1;
23.    channels[4].src_port     = WF4_Fifo_Pe4;
24.    channels[4].des_node     = ACS_HOST_NUM(0);
25.    channels[4].des_port     = 0;
26.
27.    ACS_SYSTEM  * system;
28.    ACS_System_Create(&system, nodes, 4, channels, 5, &status);
29.    ...
30.    ...
31.    ACS_System_Destroy(system);
32.    ACS_Finalize();
```

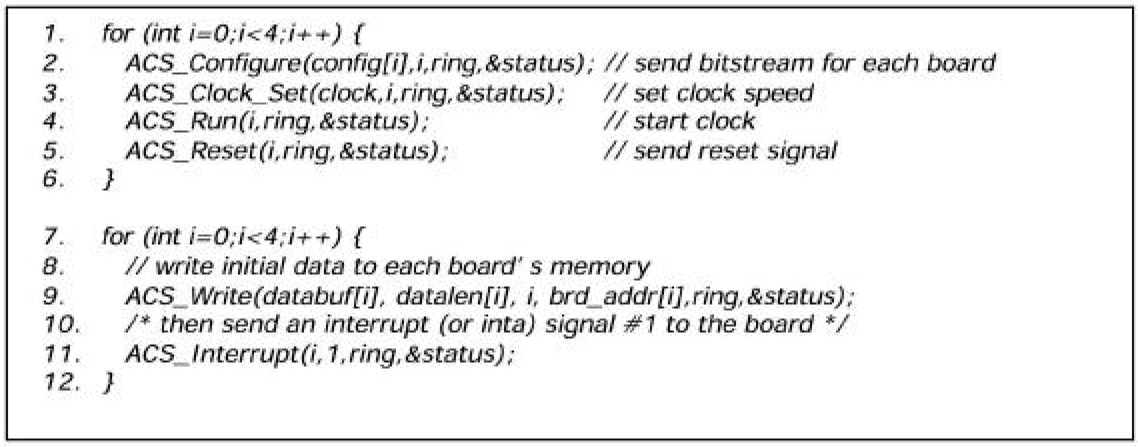
Figure 3.1: Code fragment for setting up the system

3.3 Board management

After creating the system object, the user needs to configure and control the adaptive computing board in the system. The ACS API provides routines to download and read back the configuration file to the board, set up the clock rate, send interrupt and reset signals, and start or stop running the board. Figure 3.2 shows a code fragment that uses the board management API calls to control the board.

Typically, the host program first needs to send configuration files to each board in the system. The configuration files include information to configure the processing elements and board-level configuration such as crossbar switch settings. The second line in Figure 3.2 is sending configuration files to each board, in the local and in the remote machine. After the configuration is completed, the code fragment sets the clock speed (*ACS_Clock_Set*), starts the clock (*ACS_Run*), and sends a reset signal (*ACS_Reset*) to each node in the system. Finally, as specified in Section 3.4, the code fragment uses memory access functions to write data into the board memory. Figure 3.2 also illustrates the use of *ACS_Interrupt()* to send interrupt signals to the board [11].

Although the boards are distributed in different workstations, the ACS API has completely hidden the network features from the user. To control the board, either on the local or on the remote machine, the user makes exactly the same API call.



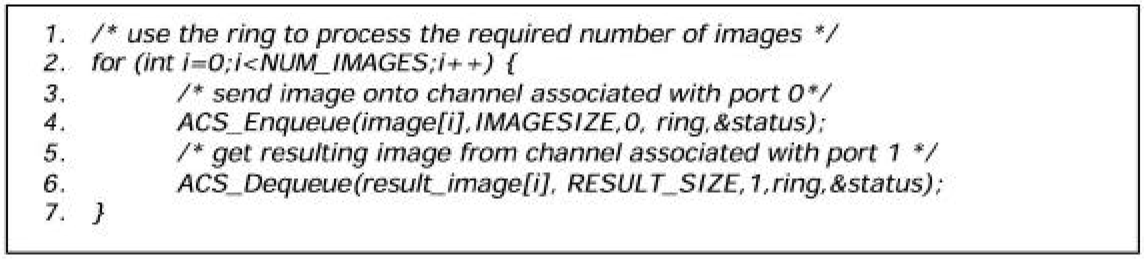
```
for (int i=0;i<4;i++) {
  ACS_Configure(config[i],i,ring,&status); // send bitstream for each board
  ACS_Clock_Set(clock,i,ring,&status);     // set clock speed
  ACS_Run(i,ring,&status);                 // start clock
  ACS_Reset(i,ring,&status);               // send reset signal
}

for (int i=0;i<4;i++) {
  // write initial data to each board' s memory
  ACS_Write(databuf[i], datalen[i], i, brd_addr[i],ring,&status);
  /* then send an interrupt (or inta) signal #1 to the board */
  ACS_Interrupt(i,1,ring,&status);
}
```

Figure 3.2: Code fragment for configuring board and writing data to the memory [11]

3.4 Memory Accessing and Data Streaming Functions

The memory access functions include *ACS_Read()* and *ACS_Write()*. They support the host computer in transferring most of the control and data to an adaptive computing device. By accessing memory-mapped data and control registers, the function can also operate a board in a local system bus.

The API supports both explicit data transfer operations, such as *ACS_Write()* and *ACS_Read()*, and implicit data transfer operations such as *ACS_Enqueue()* and *ACS_Dequeue()*. The data stream functions enable the host application to put streaming data into the system and receive streaming data from the system. In the channel-based communication model, the user only needs to explicitly specify the initial entry and final exit of the system port. The *ACS_Enqueue()* and *ACS_Dequeue()* are two primary commands for controlling the communication. Figure 3.3 describes how these commands control the data flow in a ring system. After a user calls *ACS_Enqueue()* to put data into the system, the data will go linearly through each of the boards in the system, and finally go back to the system. By adding more initializing properties when the system creates channel objects, the user can change the channel behaviors. As shown on lines 18 and 19 in Figure 3.1, the user can change the channel behaviors at creation time through specifying more detailed properties, such as the channel buffer, and the size of data being dequeued. By default, however, the user is not required to do so.



```
/* use the ring to process the required number of images */
for (int i=0;i<NUM_IMAGES;i++) {
        /* send image onto channel associated with port 0*/
        ACS_Enqueue(image[i],IMAGESIZE,0, ring,&status);
        /* get resulting image from channel associated with port 1 */
        ACS_Dequeue(result_image[i], RESULT_SIZE,1,ring,&status);
}
```

Figure 3.3: Code fragment for enqueuing and dequeuing data [11]

3.5 Non-blocking Command

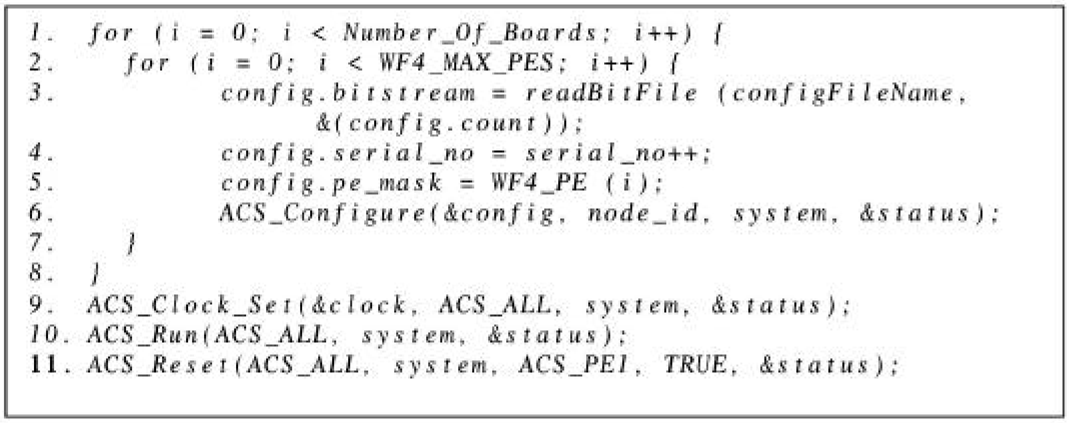
An advanced feature of the API is a set of non-blocking commands. The common API functions are blocking commands. Taking the *ACS_Write()* in Figure 3.1 as an example, the host program will be blocked when executing each write. The *ACS_Request()* function can specify a sequenc e of the API functions, which will be executed as a set. The sequence is called "a request", which can include commands such as reading/writing memory or raising a reset line.

Once initiated, the request can be committed to execution using *ACS_Commit()*. *ACS_Commit()* issues the commands, creates a handle, and returns control to the user. During the execution of these commands, the user may perform other operations. Using *ACS_Test()*, the user can easily monitor the completion of the set of commands. Using *ACS_Wait()*, on the other hand, the user can make commands wait in a blocking mode. A request may be committed to execution multiple times. The request mechanism improves efficiency by overlapping user task execution with API task execution. For example, a user can submit a non-blocking request for executing tasks in a remote node that overlaps execution on the local node. A further discussion will be given in the next chapter. By combining multiple commands and re-using command sequences, both the network and the API overhead will decrease significantly [11].

3.6 Group Operations

Functions of the ACS group management allow for creating subsets of nodes from a system with an alternative logical ordering. By using groups, the ACS API supports broadcasting and multicasting of data rather than simple point-to-point transfers. Such a broadcast function would be quite useful on a homogeneous system where all the boards are the same. For example, the programmer can configure all the boards at one time instead of calling each individually.

The *ACS_ALL* is a predefined group that represents all nodes in the system. It is used to define a broadcast operation. Figure 3.4 shows using ACS_ALL simplified the procedures for sending commands to each node. Lines 1 to 8 use loops to configure each board individually, while Lines 9 to 11 use ACS_ALL to setup the clock, run the board, and send a reset signal in just one call.



```
1.  for (i = 0; i < Number_Of_Boards; i++) {
2.     for (i = 0; i < WF4_MAX_PES; i++) {
3.             config.bitstream = readBitFile (configFileName,
                       &(config.count));
4.             config.serial_no = serial_no++;
5.             config.pe_mask = WF4_PE (i);
6.             ACS_Configure(&config, node_id, system, &status);
7.     }
8.  }
9.  ACS_Clock_Set(&clock, ACS_ALL, system, &status);
10. ACS_Run(ACS_ALL, system, &status);
11. ACS_Reset(ACS_ALL, system, ACS_PE1, TRUE, &status);
```

Figure 3.4: Using ACS_ALL to broadcast command and data to all the boards

3.7 Summary

Through the ACS API, a programmer can write a simple application that controls a complex network distributed adaptive computer system. The applications developed by the ACS API are easy to port to new systems. This chapter gave an introduction to the API and explained the rationale of the API design. The detailed specifications and descriptions of the ACS API can be viewed in Appendix A1.

4. Design and implementation

4.1 Overview

The design goal of the ACS API is to provide a simple API for the control of a complex distributed adaptive computing system. The API should also be scalable, extendible, and portable. In this chapter, there will be a detailed discussion on how the API was implemented to match with these requirements.

E machine is running a process to control the system across multiple computers. The host program, along with a control thread, controls the host machine. Other computers in the system run a *control* process, which is responsible for executing commands initiated by API calls, monitoring the local adaptive computing board, and communicating with other control processes. Each control process is single-threaded, while the host process is multi-threaded for concurrent execution of the host program. Section 4.4 gives a detailed description for the control process. Figure 4.1 illustrates how the host program and control processes control the system over the network.

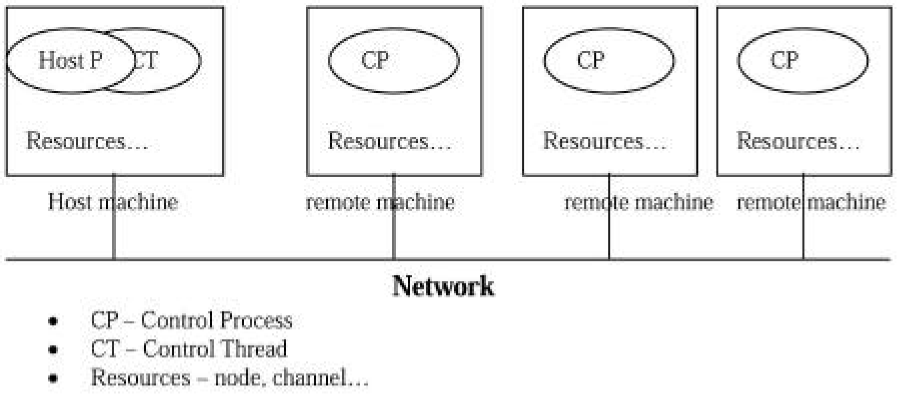


Figure 4.1: Control the ACS system over the network

Because technology is changing rapidly in the adaptive computing community, the implementation of the API requires extensibility. This goal can be achieved by using an object-oriented approach. Using objects follows naturally from the specification of the API. As outlined earlier, the *system* object is a collection of *node* and *channel* objects. A channel object represents a FIFO queue connecting one node to another node. A channel can contain a buffer and settings specifically for controlling flow on this FIFO. A node object represents a computational device in the system. Section 4.5 gives a detailed discussion about how the system object controls and manages the node and channel objects.

Two objects not directly viewed or manipulated by the user are the *communication* object and the *world* object. The *communication* object provides communication between processes on different computers. Section 4.3 gives a detailed introduction of how the ACS communication layer was built. It also discusses the buffer/memory allocation, message ordering, and flow control problems in the ACS system.

The *world* object is used to encapsulate and maintain information of the computing environment when the API is running. For example, the world object contains the network address of all node objects and channel objects. Further, it contains a list of all the adaptive computing boards managed by each control process.

The core of the API implementation is written on these objects. The classes associated with these objects, including virtual function definitions, are defined as part of the core implementation. By taking advantage of inheritance and encapsulation, the distinction between local and remote boards is easily hidden, and new types of boards and communication systems can be seamlessly included. For example, to extend the API to allow control of a new board, a developer just creates a class that inherits the node object and implements all of the virtual functions to allow for control of the new board; the rest of the API implementation remains unchanged [10]. Section 4.2 gives an introduction to the objects used in the ACS API implementation.

Section 4.6 discusses the cache configuration techniques used in run-time reconfiguration. The operating system independent layer, as part of the portability of the ACS API, is presented in Section 4.7.

4.2 Object Oriented Design

4.2.1 World object

The ACS_World object is a starting point for running an ACS system. It initializes the underlying communication, creates the control processes, and starts the performance monitor. When the program is running, it collects runtime information such as network topology, node, and channel objects location. ACS_World is a global object that shares its data with all other objects. When the program terminates, the ACS_World object releases all resources it held, and closes the underlying communication.

There is only one instance of the ACS_World object, *acs_world*. However, each machine in the network has a copy of this object. Any changes to the data in any copy of the *acs_world* will cause changes in all the others. It thus keeps the shared data identical in all the machines.

ACS_World contains the following data:

- Node location table – stores information about the node object location. The table has two columns. One is for the node' s ID; the other is where the node is located.
- Channel location table – stores information about the channel object location.
- Configurable computing board information.

ACS_World interface:

- Node registration – Each node needs to register at the world object so that all other objects can view its information. The location is important to the communication object.
- Channel registration – Same as the node registration.

Related APIs:

- *ACS_World_Info()*
- *ACS_Initialize()*
- *ACS_Finalize()*

4.2.2 System object

The ACS_System object is a collection of node objects and channel objects that have been allocated by the host application through the user-level. It manages node and channel objects. The entire system has only one instance of the ACS_System object, *acs_system*, which is on the host machine. It can manage the node and channel objects in both the local and the remote machines. There will be a discussion of the node/channel management in detail in later sections.

Figure 4.2 shows the ACS_System data, which include:

- Node array – stores all nodes – node object or virtual node object.
- Channel array – stores channel objects in the host machine. The control processes keep channels in the remote machines.
- Request array – stores all request objects.

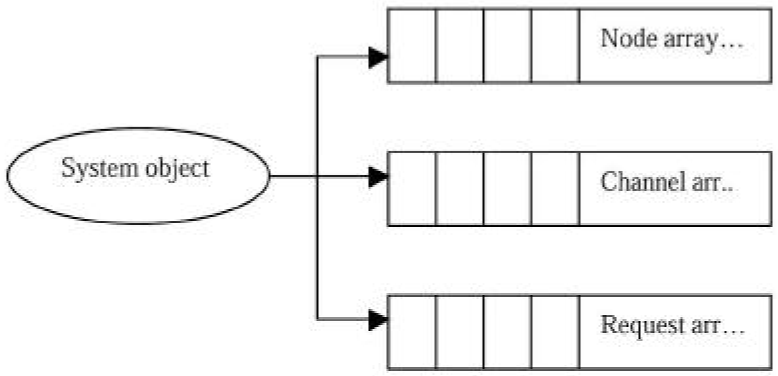


Figure 4.2: System object and its data

ACS_System interface:

- Add/Delete node object
- Add/Delete channel object
- Enqueue/Dequeue

Related APIs:

- *ACS_System_Create() / ACS_System_Destroy()*
- *ACS_Node_Add() / ACS_Node_Remove()*
- *ACS_Channel_Add() / ACS_Channel_Remove()*
- *ACS_Enqueue() / ACS_Dequeue()*

4.2.3 Node Object

Node objects represent nodes which arethe actual computing. In the ACS system, in particular, nodes refer to the adaptive computing devices that were introduced in Chapter

Two. A base node object is defined as having certain basic properties of a node; e.g., ID, board type. Other nodes can be derived from this base node. For example, consider the concept of a "local" node– a node located on the host machine. Furthermore, a WildForce node and a SLAAC node representing the Wildforce board and the SLAAC board, respectively, are derived from the local node. By inheriting the interface from the base class, accessing different nodes uses the same routines. Thus, at the API level, the user calls the function *ACS_Read()* to read data from an arbitrary board without concern for its type. This isolation of the node development from the API enables the developer to create a class that inherits the node object and implements the entire interface for control of a new board, while keeping the rest of the API implementation unchanged.

The virtual node object is introduced to represent nodes in the remote machine. The virtual node has the same interface as the local node object. It does not, however, perform the actual computation. The purpose of the virtual node is to act as a bridge to the remote node. For example, when the system object accesses the remote node, it sends commands to the virtual node. The virtual node then forwards the command to the remote node through the network. After executing the command, the remote node may reply to a message via the virtual node to the system object. By using the virtual node, accessing a distributed object becomes transparent to the higher layers; thus, making the distributed system easily managed. Moreover, it hides the underlying communication process from the API system. This feature is essential to the ACS API for transporting the implementation to other network environments. For example, the current implementation uses MPI for the underlying network communication. In the future, to use a low level

Myrinet API [17], the developer can simply replace the communication layer while the rest of the API implementation remains unchanged.

Figure 4.3 shows the Node library family' s infrastructure. By using virtual functions,

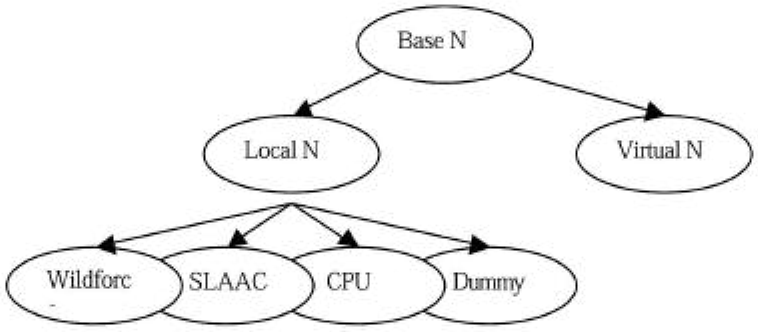


Figure 4.3Node object families

the access of different nodes is through the same interface. All nodes can recognize the same commands from the system object, even if those nodes are of different types.

Node Library

The class for each type of node has been built into a separate library. Because they inherit the same interfaces from the base node class, it is easy for a user to connect a particular type of node with the system. For example, if the Wildforce board needs to be replaced with the SLAAC board in the application, the developer only needs to declare a new board type in the program header, and link the SLAAC instead of Wldforce library with the program. The current version of ACS API includes four types of node libraries, Wildforce, WildforceF, SLAAC-1, and RCM.

The WildforceF library was revised from the Wildforce. It has enchanced the FIFO architecture to improve the FIFO throughput. The RCM board differs from other adaptive computing devices in that it uses the Context Switching Reconfigurable Computing (CSRC) chip as one of its Processing Elements. The CSRC technology was developed by Sanders, Inc. The CSRC has the ability to change between a number of programmed functions at a high-speed without adding additional FPGAs. The speed of the context switching is much faster than the speed of reconfiguration in current commercial FPGA technology. In addition, the CSRC FPGA provides the ability for sharing data between contexts, while in commercial FPGAs, the resident data is always destroyed when reprogramming the FPGA [16].

4.2.4 Channel object

A channel is a persistent communication path between two nodes or a node and a host. The endpoints of a channel are marked by the node number (or host number) and a port number. A channel object represents the channel defined above.

A channel object has the following properties:

- A pointer to the source node object along with a source port number;
- A pointer to a destination node object along with a destination port number;
- A buffer to store data that has to be transmitted to the destination;

- And properties of the channel: The size of the buffer, channel window size, which is used in flow control, and the maximum data dequeue from the source node.

A channel with a host as a source feeds data only when the host calls enqueue. Similarly, a channel with a host as a destination clears data only when the host calls dequeue. Inside the system, a channel linked by two nodes is driven by the control process to automatically exchange data. The channel object provides interfaces that can feed data into the destination node and check data from the source node. Figure 4.4 illustrates how the channel object connects the node objects in the system.

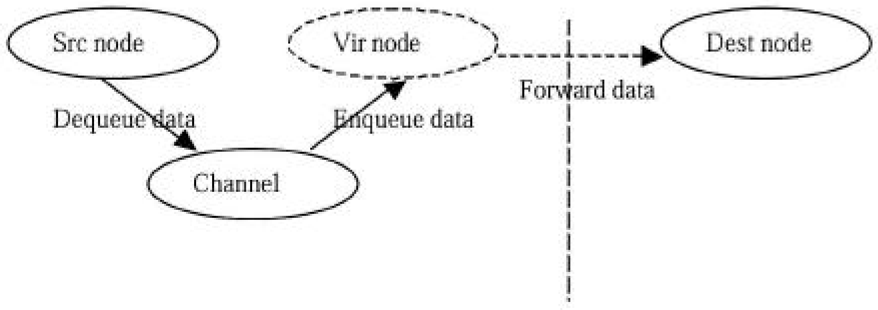


Figure 4.4: Channel object

4.2.5 Communication Object

The communication features are encapsulated into a communication object, which is in charge of all the communication procedures and provides all interfaces like sending, receiving, and broadcasting data.

The communication object:

- Establishes the communication between each other when the system starts up;
- Sends, receives, and broadcasts messages;
- Manages buffer and memory;
- And controls data stream and avoids deadlock.

The communication layer is discussed in detail in Section 4.3

4.2.6 Request object

A Request object is used for non-blocking calls. It is initialized when the user calls *ACS_Create_Request()*. After that, each time when the user calls a non-blocking function, the system will add one entry in the request object' s entry table, which includes the non-blocking function' s name, parameters, and a field for the result. After the user calls *ACS_Submit() (is this the right call, I thought it was commit, not submit???)*, the control process takes over the request object and executes the functions in the request object one by one. Because the control process exists as a thread in the host machine, the host program will not be blocked.

4.2.7 Group object

A Group object is created at the time a user calls *ACS_Create_Group()*. This object collects node objects and provides group operations. It does not, however, create any nodes. Instead, it records the nodes' IDs. For example,ACS_ALL is a global group object that represents all of the nodes in the system. It collects all of the nodes' IDs.

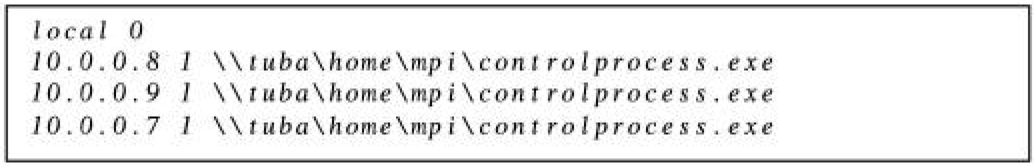
When the user wants to run the boards in all machines, instead of calling*ACS_Run()* for each of the nodes, he can simply call *ACS_Run*(ACS_ALL).

4.3 Communication layer

The communication object is responsible for all communication issues in the ACS API. The communication object uses the Message Passing Interface (MPI) for its low-level communication routines.

4.3.1 Establish communication

Recall that MPI has built-in features to support parallel computing. To invoke multiple processes among the network stations, a user needs to tell MPI the name, the number of processes, and the location of each program. MPI will then call these programs and establish the connection. The MPI program takes the command line arguments, which include the configuration file and the runtime setup.



```
local 0
10.0.0.8 1 \\tuba\home\mpi\controlprocess.exe
10.0.0.9 1 \\tuba\home\mpi\controlprocess.exe
10.0.0.7 1 \\tuba\home\mpi\controlprocess.exe
```

Figure 4.5: The configuration file used in WMPI

Figure 4.5 shows the configuration files in the Windows NT environment. The configuration files in other operating systems are similar. Each line of the configuration file consists of information for one process. The first column is the network address of a machine. Here, the network address is an IP address. The second column is the number of processes to be run on that machine. The third column is the program' s name along with the file path in that machine. In Figure 4.5, there are four machines in the system. The first is the host program, running on the host machine. The others are control processes, running on the specified machines.

4.3.2 Buffer Management

Incoming messages must be placed in the memory and passed to the control process or to the user program for processing. Meanwhile, when the control process or the user program generates output, it must be stored in the memory and passed to the communication object for transmission. The communication object accepts outgoing messages and passes incoming data to the higher layer, both in memory. The efficiency with which the communication object processes messages depends on how it manages the memory to hold the messages. A good design allocates space quickly and avoids copying data as packets move between layers.

Receiving message:

As shown in Figure 4.6, the communication object runs a typical message receiving cycle as follows:

1. Posts a receiving request to the system and waits for messages to come in;
2. Receives messages, storing them into the buffer;
3. Reads the messages and executes commands;
4. And releases the buffer and posts another receive request.

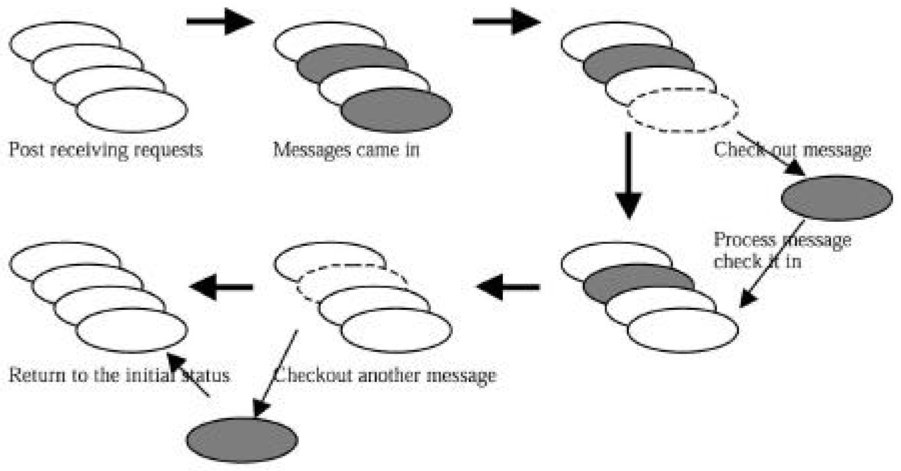


Figure 4.6: Procedures of receiving messages

When the messages come in, the communication object needs to provide buffers to store them. The fixed partitioning is a memory allocation method that lets the main memory be divided into a number of same size partitions. Each partition is for storing one message. A buffer table is used to record the buffer usage. Although it is easy to implement, the weakness is the inefficiency of using memory due to the unpredicted length of the message. For example, if 256 bytes are used for each buffer, messages of only a few bytes will result in poor memory usage.

Dynamic allocation can overcome some of the difficulties associated with fixed allocation. With dynamic allocation, each incoming message is allocated exactly the size it needs. However, the weakness is that it is difficult to maintain the fragmented memory. Extra overhead is needed when compacting the fragment memory.

To solve these problems, theACS API classifies the messages into several types based on their length. There are a total of five types of messages, which are in length between 1 to 16, 16 to 256, 256 to 4096, 4096 to 65536 and 65536 to 4M bytes respectively. Accordingly, the communication object has five types of buffers, which have 16, 256, 4096, 65536 and 4M bytes, respectively. Each type of buffer has several copies. A particular size of message can only go into a particular size ofbuffer, based on its length. As Figure 4.7 shows, if an incoming message is smaller than 16 bytes, then the communication object uses 16 bytes of buffer to receive it. If it is larger than 16 but smaller than 256, then it uses 256 bytes of buffer to receive it. This process is followed to assign messages to buffers efficiently.

An MPI tag can make the message go into a buffer of the correct size. MPI uses a tag to match a pair of sending and receiving messages. For example, the first site posts a receiving request with Tag A. If the second site wants to send a message to the first one, it must send the message with the same tag; otherwise, even if the destination is correct, the first site cannot get that message due to the mismatch of the tag. In the ACSAPI implementation, the communication object posts forty receiving requests at the same time. These forty requests are divided into five equal groups. Each group uses a particular size of buffer and tag. If one communication object is going to send a message to another, it checks the size of that message, and attaches it to the corresponding tag for that size, i.e. if the message is smaller than 16 bytes, it tags to the 16-bytes buffer; if the message is larger than 16, but smaller than 256 bytes, it tagsto the 256. Thus, only the receiver tagged for 256 will get the message.

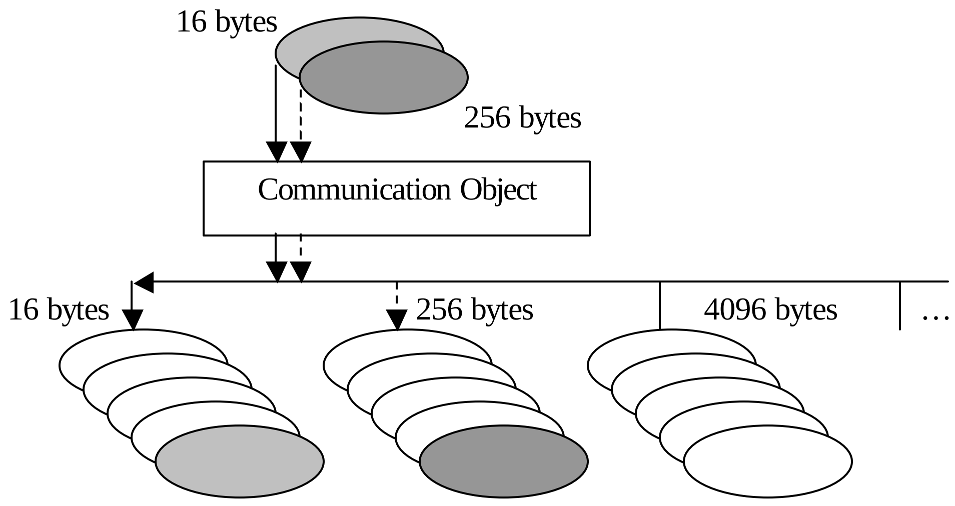


Figure 4.7: Messages will go into a right buffer size in the communication object

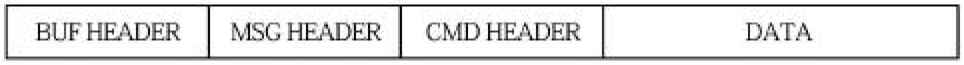
Buffer Pool data structure

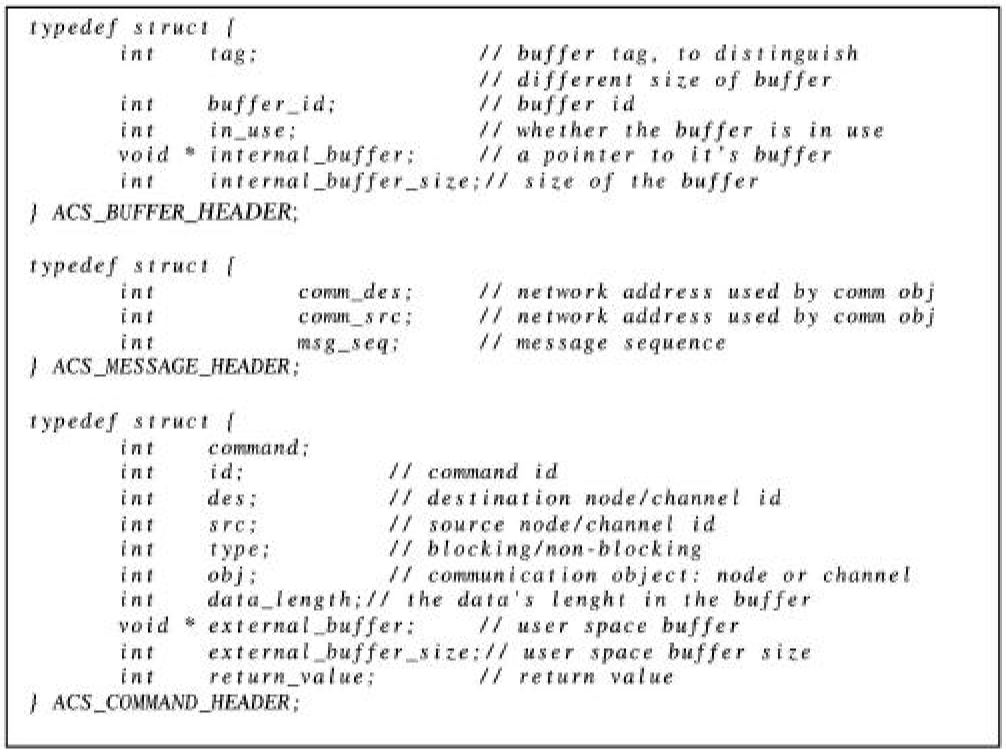
Each buffer consists of four parts: the Buffer Header, Message Header, Command Header, and the data part. The Buffer Header includes buffer ID, tag, the size of the buffer, and a field to indicate whether this buffer is in use. The Message Header includes the destination address, source address, and sequence number. The Command Header includes command ID, source/destination node ID, message type, data length, and others. Figure 4.8 below shows the buffer format and data structure.

Except for the in-use indicator, buffer header information is filled when initializing the communication object. The in-use indicator is changed each time the buffer is checking-in or checking-out. After checking out the buffer, the user program or the control process fills out the command header information. Then the buffer will be passed to the communication object. The communication object then fills out the message header and sends out the message.

Buffer Check-In and Check-Out

The buffers are recycled. After sending or receiving messages, the buffer is returned to the communication object. This is done by buffer check-in and check-out procedures. That is, when sending messages, the communication object checks out a suitable buffer size. After sending out the message, it checks the buffer back into the buffer pool. The receiving procedure is similar.





```
typedef struct {
        int     tag;                    // buffer tag, to distinguish
                                        // different size of buffer
        int     buffer_id;              // buffer id
        int     in_use;                 // whether the buffer is in use
        void *  internal_buffer;        // a pointer to it's buffer
        int     internal_buffer_size;// size of the buffer
} ACS_BUFFER_HEADER;

typedef struct {
        int             comm_des;       // network address used by comm obj
        int             comm_src;       // network address used by comm obj
        int             msg_seq;        // message sequence
} ACS_MESSAGE_HEADER;

typedef struct {
        int     command;
        int     id;             // command id
        int     des;            // destination node/channel id
        int     src;            // source node/channel id
        int     type;           // blocking/non-blocking
        int     obj;            // communication object: node or channel
        int     data_length;// the data's lenght in the buffer
        void *  external_buffer;        // user space buffer
        int     external_buffer_size;// user space buffer size
        int     return_value;           // return value
} ACS_COMMAND_HEADER;
```

Figure 4.8: A message' s format and data structure

Sending message

There are two situations to consider when sending a message.

1. Message send only – user sends commands or data to a remote node (i.e. write memory to remote node, send configuration files to remote node);
2. Message send and receive – user sends a command to a remote node to retrieve some data. This procedure consists of two steps, sending a request to a remote site and waiting for a reply (i.e. read memory from remote nodes, or a blocking call such as running a board, synchronization, etc.).

In the first case, a user or control process checks out a buffer based on the size of the command, fills out the command header, and copies the data to the buffer. It then passes the buffer to the communication object. The communication object fills out the message header,sends it out, and then checks in the buffer.

The second scenario includes two steps: sending the command to the remote site and waiting for the response back. The first step is straightforward. In the second step, after the communication object sends the message, the object posts an extra receiving request with a buffer from user space rather than from the buffer pool. It uses a different TAG with normal receiving. When the receiver sends the message back, it also uses that special TAG so that the incoming message will go directly into the user space. This saves the time of copying memory from the buffer pool to the user space. Figures 4.9 and 4.10 illustrate the buffer status changes in sending messages.

1. Check out the buffer

BUF HEADER	MSG HEADER	CMD HEADER	DATA

2. Fill out command header and data field; pass it to the communication object

BUF HEADER	MSG HEADER	CMD HEADER	DATA

3. Fill out message header; send out the message

BUF HEADER	MSG HEADER	CMD HEADER	DATA

4. Check in the buffer

BUF HEADER	MSG HEADER	CMD HEADER	DATA

Figure 4.9: The buffer status in sending messages

1. Check out the buffer

BUF HEADER	MSG HEADER	CMD HEADER	DATA

2. Fill out the command header and data field; pass it to the communication object

BUF HEADER	MSG HEADER	CMD HEADER	DATA
USER BUFFER			

3. Fill out the message header; send out the message, then wait for the response.

BUF HEADER	MSG HEADER	CMD HEADER	DATA
USER BUFFER			

4. Get the response message; Fill it in the user buffer

BUF HEADER	MSG HEADER	CMD HEADER	DATA
USER BUFFER			

5. Check in the buffer

BUF HEADER	MSG HEADER	CMD HEADER	DATA

Figure 4.10: The buffer status in sending-receiving messages

4.3.3 Flow Control

The channel is a persistent data link between two nodes. Sometimes the data rates are so fast that the transmission needs to be controlled. For example, one site sends an overwhelming number of messages to the other site. Since the communication object processes the messages one by one, unprocessed messages will have to wait in the queue.

If the sender continually sends messages, and the receiver processes messages slower than they are sent, either some messages have to be dropped, or the sender has to be blocked. Flow control is used here to control the speed of sending and receiving messages.

TCP uses window-based flow control to handle network delay, throughput, and packet loss. When the TCP process on the receiving machine sends an acknowledgement, it includes a window advertisement in the segment to tell the sender how much buffer space the receiver has available for additional data. The window advertisement always specifies the data that the receiver can accept. When the sender fills the advertised window, the value in the acknowledgement field increases and the value in the window field may become smaller until it reaches zero. TCP uses window advertisements to control the flow of data across a connection. A receiver advertises small window sizes in order to limit the data that a sender can generate. In the extreme case, advertisinga window size of zero halts transmission altogether[12].

The ACS API uses a similar concept to control the data rate between two nodes. The channel object keeps an integer called a buffer window. It represents the maximum number of messages that can be sent to the destination node. When the channel object sends a message to the destination node, the buffer window decreases by one. When the buffer window reaches zero, no further messages will be sent to that site. In the remote site, when the remote node gets a message and puts it into the FIFO, it acknowledges the sender by sending back an ACK message. Upon receiving the ACK, the channel object increases the buffer window by one. So the channel may continue sending messages to the remote node. If the remote node gets the message but cannot put the data into the FIFO, i.e. the FIFO is full, then the remote node puts the data into its channel buffer without acknowledging the sender. In such situations, if the channel object continues sending data to the remote node, the buffer window will be reduced to zero because there is no acknowledgment from the remote node. Thus, no further messages will be sent. And in the remote site, the remote node has put all the received, but not processed messages, into its buffer. The number of channel buffers in the remote node is the same as the buffer window in the channel. Conversely, when the channel buffer in the remote node is full, the buffer window in the channel object is zero and sending data is blocked until the remote node sends an acknowledgement back to the channel object. After the FIFO is empty, the remote node will move the data from its channel buffer into FIFO. It then acknowledges to the channel object, allowing more data to be sent. Figures 4.11 and 4.12 illustrate how the channel buffer is used for data flow control.

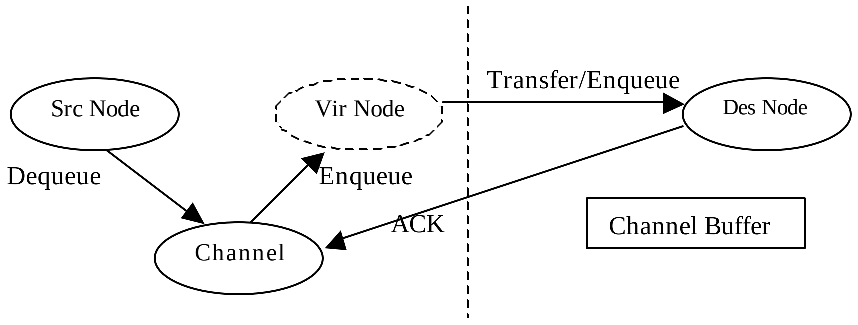


Figure 4.11: The destination node FIFO is not full and data is enqueued directly to the destination node.

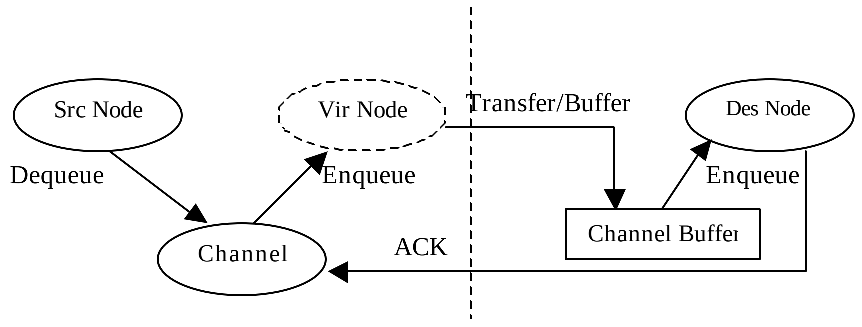


Figure 4.12: The estination node FIFO is full andata is put into the destination node' s buffer. After the FIFO is available to enqueue more data, the destination node extracts data from the channel buffer and puts it into the FIFO. Itthen acknowledges this to the channel.

4.3.4 Message ordering and serialization

Because the communication object will receive multiple messages at the same time, it is important to process the message in the sequence of messages sent. For example, one sends out two messages for setting the frequency of the board and then for running it. If the communication object executes the run board command before setting the board up, an error will occur.

The API uses a sequence number in each message' s header to indicatethe order in which the message is received. The communication object keeps a table for each site to record what the next message' s sequence is from that site. If an incoming message has an unexpected sequence number, the communication object will suspend it and wait for the correct message.

Because each site keeps its own sequence number table, there is no global sequence number. Although using individual sequence numbers can keep the messages from the same site in sequence, it cannot guarantee that themessages sent by different sources keep their original order. For instance, when site A and site B send two commands respectively to site C, there will be a total of six possible sequences in which site C may receive these messages.

Fortunately, the ACS API is a simple parallel application. When the program is running, most of the network traffic is for passing channel data. Because the FIFO in the reconfigurable board is usually designed to process a single source of data, one assumes that each FIFO in a node will not be linked by more than two channels. Thus, although a board can have multiple channels to connect, receiving messages from different channels will not cause any conflicts. The messages can be distinguished by the FIFO name.

Another situation is when the host machine sends commands to other machines. That typically occurs during the setup or configuration time. The host machine will communicate with multiple machines at the same time. However, most of the messages are send-only. That means the host machine sends messages without acknowledgement. Therefore, no machine in the system will receive messages from more than two sources at the same time. Those messages which require a reply, such as *ACS_Read()* and *ACS_Run()*,are sent in a blocking fashion. That means the host program will not send further messages until it gets a reply from the machine to which it sent the message. Thus, all acknowledged messages received in the host machine are serializedand do not conflict with each other.

4.4 Control Process

The Control Process is not accessible by the user. Still it is an integral part of the ACS API. The control process runs on each PC and is responsible for all nodes and channels on that PC. It accepts commands from the host program or other controlprocesses to control the nodes.

As Figure 4.13 shows, the control process runs an infinite loop to perform various tasks continually until the program is terminated. It checks the incoming messages, interprets them, and manipulates the board based on the commands. It drives the channel to read data from the source node and send it to the destination node, and maintains a request object.

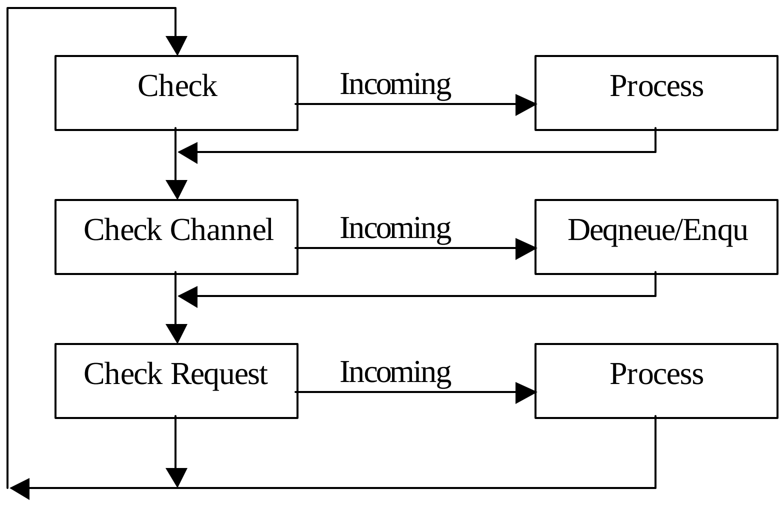


Figure 4.13: Control Process Loop

4.5 Node/Channel Management

Node and channel objects are managed by the system object and the control process. Both the system object and control process maintain two arrays: one is for nodes and the other is for channels.

To add a node into the system object, the user needs to call *ACS_System_Create()* or *ACS_AddNode()*. The node information is specified in the ACS_NODE data structure. Upon receiving a request, the system will register the node to the world object. Because the world object is a global object, the information about the node is broadcast to all sites. Therefore, all of the control processes know where the node resides.

If the node is on the host machine, the system object creates a node object and puts it into its node array. If the node is on the remote machine, then the system object creates a virtual node object in its node array and sends a command to the corresponding control process on which that node should reside. That control process then adds a node object to its node array.

To add a channel into the system object, the user needs to call *ACS_System_Create()* or *ACS_AddChannel()*. The channel information is specified in the ACS_CHANNEL data structure, which includes source node ID, the source port, destination node ID, destination port, and other properties such as the maximum size of data to dequeue from the FIFO and the channel buffer size in ACS_CHANNEL. If the user does not specify the channel information, a default value will be assigned. Similar to adding a node object, the channel object needs to be registered at the world object so that all control processes will know where the channel resides.

Based on the type of source and destination, the channel has three types:

- **Host to node –** This type of channel is used as the beginning of a channel link. The source endpoint of the channel is a "host port." When the user puts data into that "port," it will go through that channel. The system may have several of these types of channels so as to provide multiple channel links. This type of channel is only used in the host machine.
- **Node to node –** This type of channel links two nodes. The channel object is connected with one node object as a source node and a virtual node object as the destination node.

  If the source node is on the remote machine and the destination node is also on the remote machine, the channel object needs to create a virtual node to represent that destination node. In this case, although the host machine already has a virtual node object for that destination node, the system needs to create another virtual node object in the channel source node machine. Thus, there will be more than one virtual node object linking to one node object in the system. Figure 4.14 shows this situation.

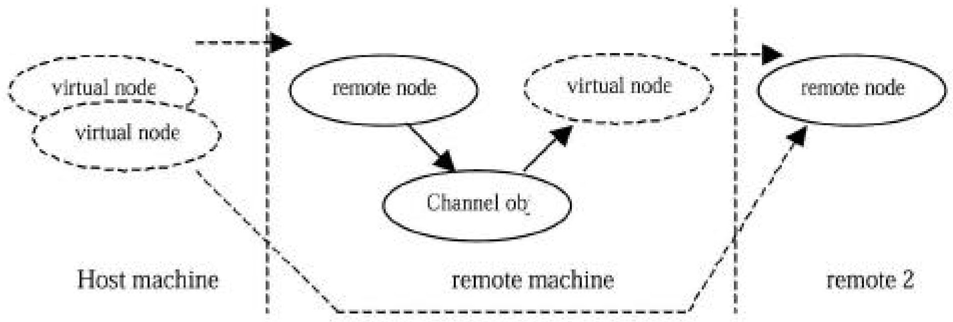


Figure 4.14: The remote node on the remote machine two
has two copies of virtual node objects.

- **Node to Host** – This type of channel is used for the endpoint of a channel link. When users create such a channel and point it to a host "port," the system will create a dummy node automatically. That dummy node is used as an extra buffer to receive returned data. Therefore, a channel linking a node and a host is actually between a computing node and a dummy node. Figure 4.15 shows this.

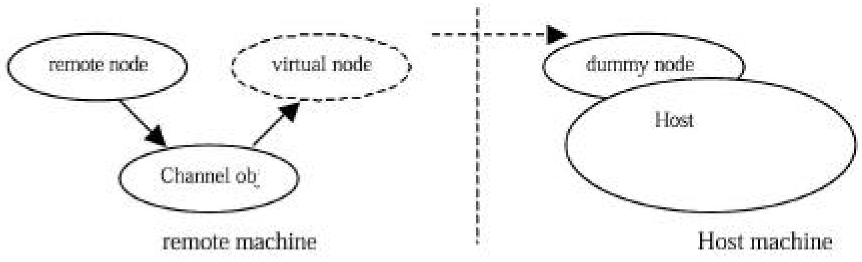


Figure 4.15: This type of channel is from a node to a host.

<u>Deleting a Channel</u>

If the channel to be deleted is in the host machine, then the system object will remove it from the channel array. If the channel is on a remote machine, then the system object sends a command to the corresponding control process. That control process removes the channel object from its array and deletes the virtual node object linked to it.

4.6 Cache Management & Run-time Reconfiguration

In some cases, the node (FPGA board) needs to be configured at run-time. The user will send different configuration files to the board in order to change its functionality. Often, the usage pattern of configuration files demonstrates temporal locality. To reduce the network transmission time, the configuration files are cached in the remote site. If on the second occasion, the user tries to send old configuration files, the system will just send the configuration file' s header rather than the data to the remote site.

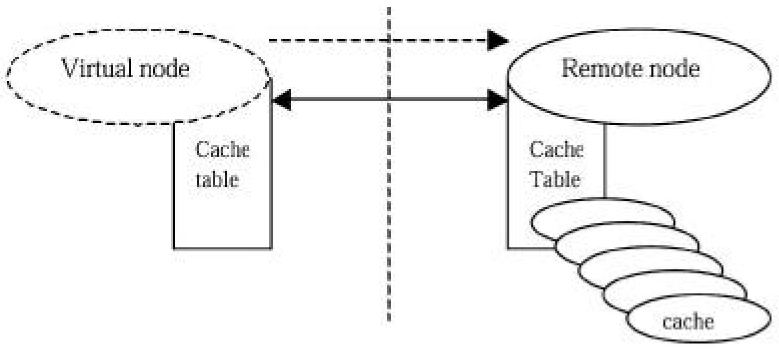


Figure 4.16: Cache configuration file in the remote machine to reduce the network transmission

As Figure 4.16 above shows, the virtual node in the local site and the node in the remote site both maintain a cache table. The two tables are identical in content, but only the remote node caches the actual data. When a new configuration file comes in, the virtual node searches its cache table to see if this configuration file has been cached. If it has, the virtual node will not send any actual data but only the configuration file head. In the remote site, if the node gets a configuration message, it checks the configuration file' s ID with those in the cache. Because the two cache tables are identical, the remote node will get the cached data in its buffer. If the virtual node did not find the configuration file' s ID in its table, by default it uses a replacement algorithm to put the configuration files' header in its cache table. It then sends out the whole configuration file. The remote node then uses the same replacing algorithm to save both the configuration file' s header and the data into the cache.

Explicit and implicit control of the cache

The user is responsible for the selection of a configuration ID. If a user sends two different configuration files with the same ID, the system will consider them as the same. The user may have the option to explicitly control the cache by specifying the cache slot in the configuration file' s header.

4.7 OS Independent Layer

The ACS API was written in standard C/C++ language. Most of the code can be compiled directly under the Windows NT or Linux environment. However, some sections of the code depend on different operating systems. For example, Windows NT uses *_beginthread* to create a thread. While in Linux, the function is *_pthread_create*. Using the critical section is also different in the two operating systems.

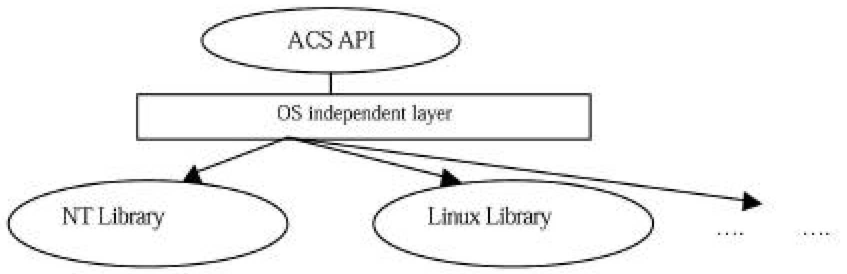


Figure 4.17: OS Independent layer

An OS independent layer has been inserted between the API and the system call. As Figure 4.17 shows, instead of calling the system routine directly, the ACS API calls the OS independent routine. For example, the function *_createthread* takes the place of *_beginthread or _pthread_create*. The implementation of *_createthread* varies by operating system.

Through the OS independent layer, the source code of ACS API can remain unchanged. To change an operating system, the developer just needs to provide the implementation of the OS independent layer.

4.8 Summary

This chapter discussed the detailed implementation of the API, and told the user how the API was implemented to be scalable, portable, and extendible. Appendix A3 shows the objects class header files used in the API. The next chapter will discuss how to test the API and analysis the API performance.

5. Testing APIs

5.1 Overview

The ACS API has been tested and analyzed by running various test applications on the Virginia Tech Tower of Power platform. This chapter describes the test procedures and analyzes the API performance. The API has been tested under three different environments: Windows NT with 100 Mbps switched Ethernet, Linux with 100 Mbps switched Ethernet, and Linux with 1.2 Gbps switched Myrinet. All tests used the Wildforce board. The version of MPI used for Windows NT is WMPI 1.2, MPICH 1.2.0 for Linux with switched Ethernet, and MPICH_GM 1.1.2.13 for Linux with Myrinet. Based on the API functions, the test programs have been divided into five categories. Section 5.2 discusses the testing programs and analyzes the performance of the API. Section 5.3 categorizes all the API functions that were involved in the test programs into a table.

5.2 Test Programs and Performance Results

5.2.1 Memory Access Function Test

The program listed in Appendix A2.1 tests the memory access functions, *ACS_Read()* and *ACS_Write()*. The program tests the memory access functions both in local and remote machines. The program calls *ACS_Write()* to write various sizes of data into the board, local and remote. It then reads data back from the boards through *ACS_Read()*. After that, the program compares the data written with the data that was read to ensure the operation was performed correctly. Tables 5.1 and 5.2 show the average time for accessing 4 bytes to 1 Mega-byte of data in local and remote boards. Table 5.1 gives the results from Linux environment, and Table 5.2 from Windows NT. The memory-access rate and network data transfer rate are calculated by the equations below. Because the memory accessing includes both reading and writing, the rate is doubled.

$$Access\_Rate = \frac{Data\_Size \times 2}{Access\_Time}$$

$$Data\_Transfer\_Rate = \frac{Data\_Size \times 2}{(Remote\_Board\_Access\_Time - Local\_Board\_Access\_Time)}$$

The average memory-access rate for the local board in the Linux environment is about 9.3 MB/s. The access rate for the remote board is 3.7 MB/s in the switched Ethernet, and 5.4 MB/s in the Myrinet. The difference is primarily from the overhead of transferring data through the network. Other overheads are from the latencies in reading and writing data into the memory. When the data size is very small, these overheads are very apparent. As shown in Table 5.1, the access rate is only 0.031 MB/s for the local board and even less for the remote board when the data size is 4 bytes. The results also showed that using a high-speed network could significantly improve the system performance.

The results from the Windows NT with switched Ethernet environment are much better than in the Linux. The average memory-access rate for the local board is about 64 MB/s, which is almost six times larger than in the Linux environment. The average memory-access rate for the remote board is about 6.4 MB/s, which is also larger than in the Linux environment.

Table 5.1: Average memory access time in Linux environment

Data Size (bytes)	Local Board		Remote Board in 100Mbps Switched Ethernet		Remote Board in 1.2Gbps Switched Myrinet	
	Access Time (s)	Access Rate (MB/s)	Access Time (s)	Access Rate (MB/s)	Access Time (s)	Access Rate (MB/s)
4	0.000251	0.031872	0.042591	0.000186	0.016468	0.000484
132	0.011041	0.023910	0.013815	0.019110	0.039479	0.006686
260	0.009086	0.057230	0.011905	0.043676	0.020014	0.025980
388	0.008062	0.096258	0.010853	0.071500	0.001508	0.514480
516	0.006130	0.168352	0.010906	0.094628	0.001440	0.716662
644	0.004196	0.306980	0.012923	0.099666	0.001465	0.879178
772	0.002276	0.678282	0.011033	0.139938	0.001476	1.046062
900	0.000357	5.037314	0.011109	0.162030	0.001479	1.216752
102400	0.016783	12.202824	0.052144	3.927610	0.038432	5.328898
204800	0.028370	14.437786	0.094570	4.331168	0.077488	5.285978
307200	0.044703	13.743942	0.165127	3.720764	0.114831	5.350458
409600	0.068220	12.008268	0.230258	3.557744	0.147832	5.541438
512000	0.112960	9.065182	0.284365	3.601006	0.194632	5.261220
614400	0.114396	10.741666	0.349091	3.520002	0.204280	6.015262
716800	0.171892	8.340120	0.391599	3.660888	0.255927	5.601604
819200	0.179795	9.112618	0.458176	3.575916	0.301366	5.436586
921600	0.214502	8.592926	0.494910	3.724312	0.340540	5.412576
1024000	0.220141	9.303114	0.556146	3.682484	0.379426	5.397632

One explanation for the difference is that the network transfer rate in Windows NT is higher than in Linux. However, examination of the actual network data transfer rates through the equation above reveals that the data rate under Windows NT is close to the data rate under Linux with switched Ethernet but even slower than the Linux with Myrinet. The network data transfer rate under Linux with switched Ethernet is 6.1 MB/s, 12.8 MB/s under Myrinet, and 7.8 MB/s under Windows NT with switched Ethernet. The correct explanation is the DMA read and write in Linux is much slower than in Windows NT. This happened in both reading/writing memory and in the FIFO operation. Thus, although the network data transfer rate is the Myrinet is high, the Linux environment has low performance.

Table 5.2: Average memory accessing time in Windows NT environment

Data Size (bytes)	Local Board		Remote Board	
	Access Time (s)	Throughput (MB/s)	Access Time (s)	Throughput (MB/s)
4	0.000630	0.012680	0.003782	0.002114
132	0.000342	0.771160	0.002397	0.110122
260	0.000329	1.580540	0.018572	0.027998
388	0.000329	2.356260	0.002470	0.314126
516	0.000332	3.111540	0.002392	0.431438
644	0.000332	3.879500	0.002485	0.518308
772	0.000326	4.736180	0.002384	0.647560
900	0.000396	4.541620	0.002376	0.757574
102400	0.003398	60.276661	0.041407	4.945984
204800	0.006513	62.892818	0.083702	4.893550
307200	0.009832	62.491940	0.103384	5.942874
409600	0.013147	62.310799	0.117353	6.980628
512000	0.016134	63.468460	0.158898	6.444400
614400	0.020586	59.691059	0.198057	6.204284
716800	0.021722	65.997597	0.207016	6.925058
819200	0.024627	66.528603	0.245347	6.677888
921600	0.027513	66.992981	0.273556	6.737934
1024000	0.030177	67.866997	0.291504	7.025624

5.2.2 Stream Data Function Test

The stream data test programs listed in Appendix A2.2 test the *ACS_Enqueue()* and *ACS_Dequeue()* functions. They also test the board management functions, including *ACS_Configure()*, *ACS_Reset()* and *ACS_Run()*. Like the steps shown in Figure 3.2 and 3.3, the test program links the nodes into a ring and uses board management functions to configure the board, start the clock, and send the RESET signal to the board. It then calls *ACS_Enqueue()* to put data into the system object. The data should pass through each board in the ring automatically and finally go back to the system object. The test program then calls *ACS_Dequeue()* to get the data back from the system object.

The performance of the data stream functions is measured in two ways by this program: the round trip time (RTT) and the throughput The RTT is the total time for putting data into the system, passing through and processing data in each board, and getting data back from the system. Obviously, the RTT varies by the number of the boards in the system. The program in Appendix A2.2a tests the RTT in one to eight machines in both the Windows NT and Linux environments. The RTT values are listed in Table 5.3. They are based on the average of 1000 repetitions of transmitting 1024 bytes data through the system.

Table 5.3: Average round trip time to pass through 1024 bytes data to system

Number of Boards	Average RTT in Win NT with Ethernet	Average RTT in Linux with Ethernet	Average RTT in Linux with Myrinet
1	0.00038	0.002255	0.002490
2	0.00284	0.020193	0.004389
3	0.00414	0.034057	0.011979
4	0.00953	0.052175	0.015276
5	0.01083	0.070366	0.021949
6	0.01355	0.097696	0.023934
7	0.01688	0.123697	--
8	0.02220	0.154830	--

Taking the results from Windows NT, as shown in Table 5.3, the RTT was increased approximately by 0.002 ~ 0.004 seconds for each additional board in the system. The results from the Linux also show a similar linear increasing. Because there is no network transaction in a one-board system, the RTT is much lower than in the others. Table 5.3 shows a big increase in time from a one-board to a two-board system. The RTT in Linux is about five times larger than in Windows NT as shown in the first line of Table 5.3. This is because of the low efficiency of the DMA access in the Linux environment. This affected the entire system as shown in Figure 5.1, the slope of Linux with switched

Ethernet is much steeper than the Windows NT. However, with the compensation from the high-speed network, the curve for Linux with Myrinet is close to that for Windows NT.

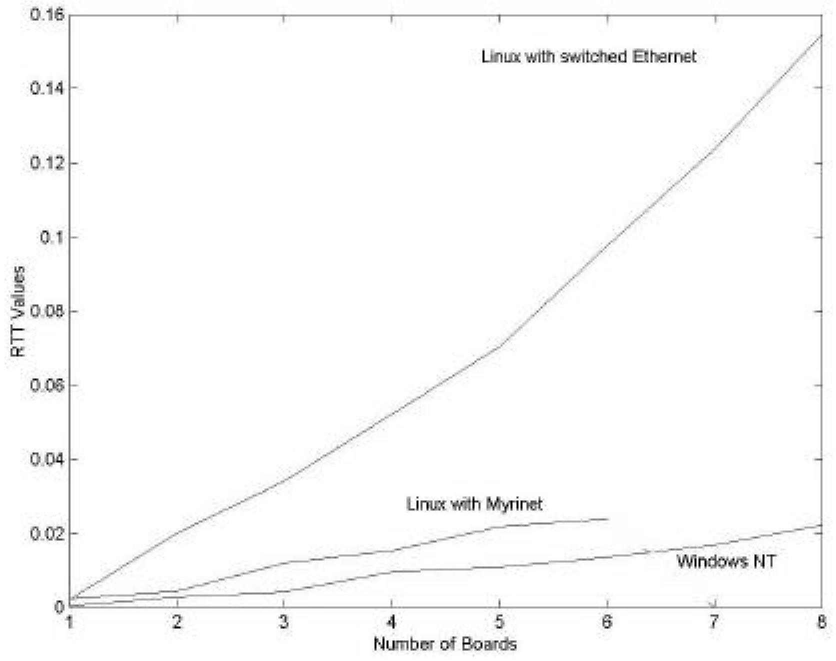


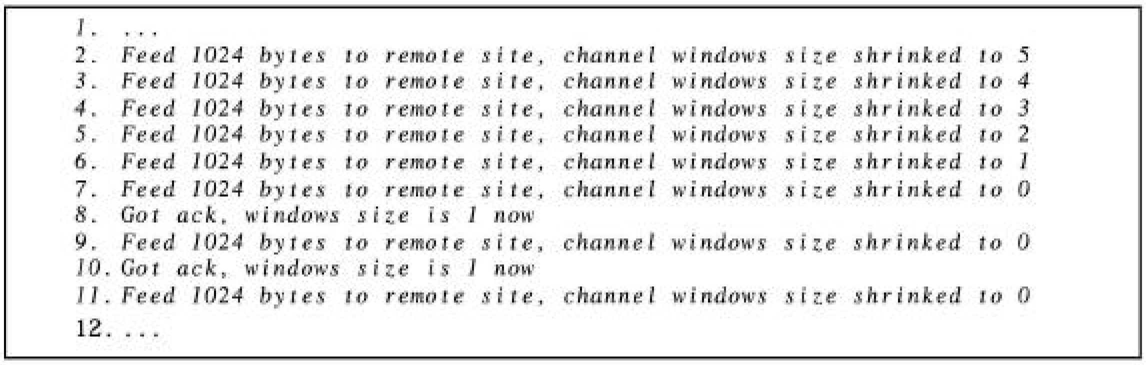
Figure 5.1: RTT Values with the number of boards in the system

Appendix A2.2b contains the program for testing system throughput. It differs from the RTT test program in that it creates a thread to dequeue the data instead of calling *ACS_Dequeue()* in the host program. The host program then enqueues five megabytes of data into the system and records the total time elapsed. Table 5.4 lists the average system throughput tested in the three enviromments. As seen from the results, the throughput for Windows NT and for Linux with Myrinet remains constant as the network size increases. This is essential to the scalability of the ACS system. That means the user can increase the power of the ACS system by adding more processing elements while at the same time keeping the same data processing speed.

Table 5.4: System Average Throughput in different number of boards

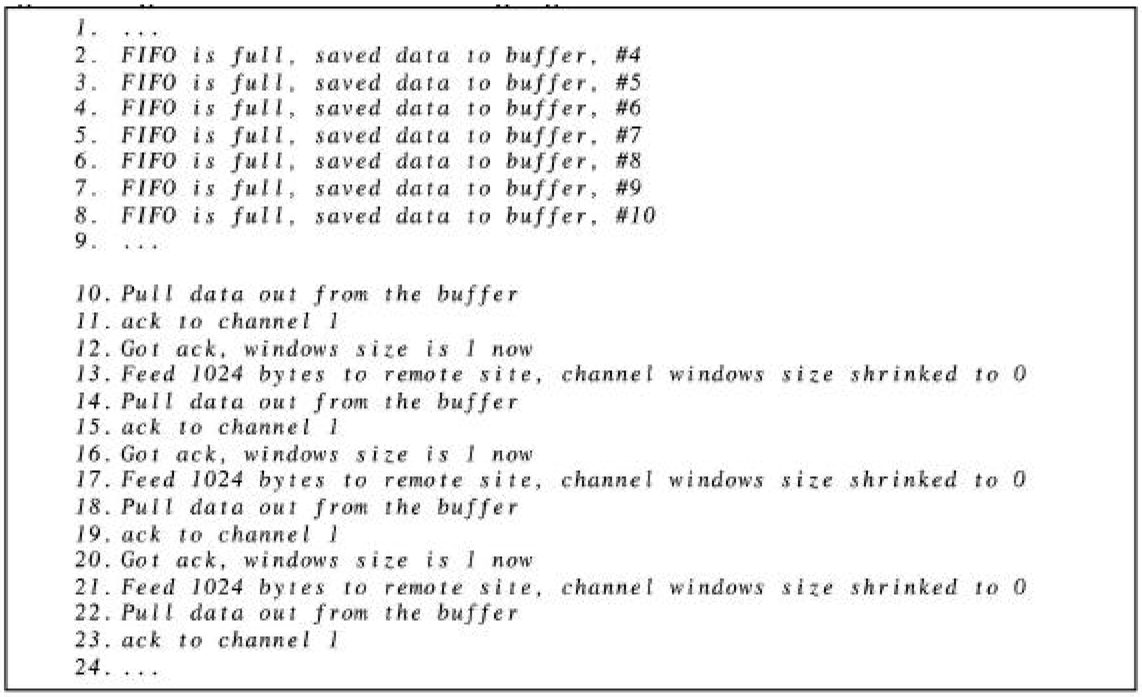
Number of Boards	Average Throughput In Windows NT with Ethernet	Average Throughput In Linux with Ethernet	Average Throughput In Linux with Myrinet
1	$2.5 * 10^6$	191242	227607
2	563834	71088	220951
3	567826	45079	224177
4	547496	33253	228139
5	575025	28865	224794
6	572278	22955	--
7	622225	23440	--
8	586686	18163	--

When testing system throughput, the data rate in the system is fast because the data continues to pass from one node to another. The log file in this test shows that the flow control effectively controlled the data rate. When data generates too quickly from the first node, as shown in Figure 5.1, the channel window rapidly drops to zero, thus blocking the first node. Meanwhile, in the second node, since the FIFO is full, the data has been saved into the channel buffer. After FIFO is available, as shown in Figure 5.2, line 10, it pulls data out from the buffer and puts it into the FIFO. It then acknowledges it to the channel in the host machine (Figure 5.2, line 11). In the first machine, when the channel gets the acknowledgement, it increases the channel window by one and continues to send data to the second node (Figure 5.1, lines 8 to 9).



```
1.  ...
2.  Feed 1024 bytes to remote site, channel windows size shrinked to 5
3.  Feed 1024 bytes to remote site, channel windows size shrinked to 4
4.  Feed 1024 bytes to remote site, channel windows size shrinked to 3
5.  Feed 1024 bytes to remote site, channel windows size shrinked to 2
6.  Feed 1024 bytes to remote site, channel windows size shrinked to 1
7.  Feed 1024 bytes to remote site, channel windows size shrinked to 0
8.  Got ack, windows size is 1 now
9.  Feed 1024 bytes to remote site, channel windows size shrinked to 0
10. Got ack, windows size is 1 now
11. Feed 1024 bytes to remote site, channel windows size shrinked to 0
12. ...
```

Figure 5.2: Log file in host machine when testing large stream data



```
1.  ...
2.  FIFO is full, saved data to buffer, #4
3.  FIFO is full, saved data to buffer, #5
4.  FIFO is full, saved data to buffer, #6
5.  FIFO is full, saved data to buffer, #7
6.  FIFO is full, saved data to buffer, #8
7.  FIFO is full, saved data to buffer, #9
8.  FIFO is full, saved data to buffer, #10
9.  ...

10. Pull data out from the buffer
11. ack to channel 1
12. Got ack, windows size is 1 now
13. Feed 1024 bytes to remote site, channel windows size shrinked to 0
14. Pull data out from the buffer
15. ack to channel 1
16. Got ack, windows size is 1 now
17. Feed 1024 bytes to remote site, channel windows size shrinked to 0
18. Pull data out from the buffer
19. ack to channel 1
20. Got ack, windows size is 1 now
21. Feed 1024 bytes to remote site, channel windows size shrinked to 0
22. Pull data out from the buffer
23. ack to channel 1
24. ...
```

Figure 5.3: Log file in remote machine when testing large stream data

5.2.3 Non-blocking Function Testing

The non-blocking test program in Appendix A2.3 is revised from the memory access test program. The first part of the program uses normal blocking functions *ACS_Write()* and *ACS_Read()* for accessing the memory in the local and remote boards. The second part illustrates how the user takes advantage of using non-blocking functions to reduce the total execution time by overlapping the computation and communication. In that part, the program first calls the non-blocking functions for accessing the memory in the remote board. It then calls *ACS_Commit()* to execute the requests in the background. After that, the program uses the blocking functions for accessing the memory in the local board and waits for the non-blocking request to finish. Because accessing the memory in the remote board is executed concurrently with the access in the local board, the time for transferring data through the network is overlapped with the time for writing data into the local board. Comparing the first part, which uses blocking functions for accessing both the local and remote boards, the total execution time is reduced.

Because of the frequent context switching in the concurrent execution, using non-blocking functions introduces some extra overhead. The use of a large number of non-blocking calls should be carefully considered to ensure that the extra overhead does not counteract the time saved from concurrent execution. As the last line in Windows NT column in Table 5.5 shows, when more and more non-blocking functions were called, the total execution time became worse than if blocking functions were used.

Table 5.5: Using non-blocking functions reduces the total execution time.

Times of Accessing Memory in Remote Board	Windows NT with Switched Ethernet		Linux with Myrinet	
	Total Execution Time When Using Blocking Function	Total Execution Time When Using Non-blocking Function	Total Execution Time When Using Blocking Function	Total Execution Time When Using Non-blocking Function
5	3.222588	2.245843	12.492957	10.630059
10	4.662546	3.876939	14.655513	10.941236
15	6.278506	6.026721	16.395372	10.967671
20	7.731978	8.663711	18.280171	11.356604

Comparing the results from the NT, the total execution time with non-blocking functions in Linux did not increase significantly as the number of remote memory accesses increased. The total execution time with blocking calls in Linux did increase significantly. The reason is because the Linux version takes more time to access memory than the Windows NT version and the time used for accessing memory is much larger than the time for transferring data through network. Thus, the time for accessing remote memory was entirely overlapped with accessing the local board in the Linux environment.

5.2.4 Group Function Testing

This program is also revised from the data stream test program. Instead of using an individual node ID in calling the board management functions, it uses ACS_ALL, so the user only needs to call those functions once. The program is listed in Appendix A2.4.

5.2.5 Configuration Cache Testing

In this test, the host program sends three different configuration files to the remote node repeatedly. Since the configuration files have been cached in the remote machine the first time it was configured, the system will not send any actual configuration data to the remote machine in the later *ACS_Configure()* calls. As shown in Table 5.6, the times for reconfiguring the remote boards have been significantly reduced after the first call. The program is listed in Appendix A2.5.

Table 5.6: Time of reconfiguring board

Board Number	Time of first configuration	Time of second configuration
1	--	--
2	0.140349	0.031716
3	0.136511	0.031293
4	0.381436	0.032106

5.3 Test Table

All the APIs have been successfully tested. The Test Table shows the APIs involved in the above test programs.

Table 5.7: API Test Table

	Memory Access	Data Stream	Non-Block	Group Function	Config Cache	Other
Support Functions						
ACS_Version	X	X	X	X	X	
ACS_Initialize_Log					X	
ACS_World_Info	X	X	X	X	X	
System Setup Function						
ACS_Initialize	X	X	X	X	X	X
ACS_Finalize	X	X	X	X	X	X
ACS_System_Create	X	X	X	X	X	X
ACS_System_Destroy	X	X	X	X	X	X
ACS_Channel_Add						X
ACS_Node_Add						X
ACS_Is_Local						X
ACS_Is_Remote						X
Memory Access Function						
ACS_Read	X					
ACS_Write	X					

Board Management Function

ACS_Configure		X		X	X	
ACS_Readback						X
ACS_Clock_Set		X		X	X	
ACS_Clock_Get						X
ACS_Run		X		X	X	
ACS_Stop						X
ACS_Reset		X		X	X	
ACS_Reset_Toggle						X
ACS_Interrupt						X
ACS_InterruptPoll						X
ACS_Reg_Read						X
ACS_Reg_Write						X

Data Stream Function

ACS_Enqueue		X	X	X	X	
ACS_Dequeue		X	X	X	X	

Group Access Function

ACS_Group_Create				X		
ACS_Group_Destroy				X		

Non-blocking Function

ACS_Request_Create			X			
ACS_Request_Destroy			X			
ACS_Commit			X			
ACS_Wait			X			
ACS_Test			X			
ACS_ReadN			X			
ACS_WriteN			X			
ACS_ResetN			X			
ACS_ConfigureN			X			

6. Conclusion and Future Work

6.1 Summary

In this research, a scalable API and runtime software system was designed and developed to support applications of the network distributed ACS system. This thesis work discussed in detail the design and implementation of the API, as well as the testing and analysis of the performance of the API implementation.

The latest version of the ACS API is 1.2. This release was tested and released on January 2000. It supports the SLAAC-1, SLAAC-2, RCM, and Wildforce boards. It uses MPI 1.1 for interprocessor communication. This API implementation has been tested in the Linux and Windows NT environments. The API specifications, implementation source code, and supporting documents can be accessed on Virginia Tech Configurable Computing lab web site at http://www.ccm.ece.vt.edu/slaac/.

6.2 Future Work

One of the issues mentioned in the previous chapters is that the control process takes a significant amount of CPU time to check the messages from the network card and the data from the computing devices. Compared with the CPU time, the message and data rates are very slow. Thus, most of the CPU time has been wasted on checking messages and data. The next step of the API implementation is to integrate the control process into the network and reconfigurable computing cards. Although the API and the host program still need support from the operating system, the control process no longer needs support from the operating system and requires no CPU time.

Efficient support for run-time reconfiguration (RTR) in the current version of the API allows for the control processes to cache several configurations to reduce the network transfer requirements. This mode is called host-initiated RTR because the host process is always the initiator of reconfiguration. The next step is to develop a more complex data-driven RTR, in which the reconfiguration is driven by the data that is encountered. The host process cannot drive such reconfiguration efficiently.

Another step of the ACS API development is to integrate the existing version with other high-level tools. For example, JHDL provides a portable, integrated environment for programming a single adaptive computing board [7]. With JHDL, a user can write or debug applications on the FPGA board through a high-level host program. Because JHDL uses Java as its programming environment, it could be integrated with the ACS API. Such integration can further simplify the steps of writing applications in distributed adaptive computing systems [11].

References

[1] D.Ridge, D.Becker, P.Merkey, and T.Sterling, "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs," *in Proceedings, IEEE Aerospace*, 1997.

[2] Virginia Tech Configurable Computing Lab, "Tower of Power," *http://www.ccm.ece.vt.edu/slaac/,* 1998

[3] Annapolis Micro System, Inc., "Wildforce Reference Manual," *Annapolis Micro System, Inc.,* 1997

[4] Myrinet, Inc., "Myrinet Reference Manual," *Myrinet, Inc.,*

[5] Xilinx, Inc., "The Programmable Logic Data Book," *Xilinx Inc.,* 1993.

[6] J.R.Armstrong and F.G.Gray, "Structured Logic Design with VHDL," *Prentice Hall*, 1993.

[7] P. Bellows and B. Hutchings, "JHDL-An HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machine*s, April 1998.

[8] Information Sciences Institute – East, "Systems Level Applications of Adaptive Computing", *http://www.east.isi.edu/SLAAC/,* 1998.

[9] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputing Applications, Vol. 8(3/4),* 1994.

[10] Allison L. Walters, "A Scaleable FIR Filter Implementation Using 32-bit Floating-Point Complex Arithmetic on a FPGA Based Custom Computing Platform", *Virginia Tech Electronic Thesis and Dissertation.* 1998

[11] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, "Implementing an API for Distributed Adaptive Computing Systems," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA*, April, 1999.

[12] Duncan A. Buell, Jeffrey M. Arnold, Walter J. Kleinfelder, "Splash 2, FPGAs in a Custom Computing Machine", *IEEE Computer Society Press,* 1996

[13] Douglas E. Comer, David L. Stevens, "Internetworking with TCP/IP, Volume II, Design, Implementation, and Internals", *Prentice Hall,* 1994

[14] B. Schott, S. Crago, R. Parker, L. Carter, C. Chen, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Reconfigurable Architectures for System-Level Applications of Adaptive Computing," *submitted to VLSI Design special issue on reconfigurable computing.*

[15] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, "MPI: The Complete Reference", *The MIT Press*, 1997.

[16] Stephen M.Scalera, Jose R. Vazquez, "The Design and Implementation of a Context Switching FPGA", *FCCM*, 1998.

Appendix 1. The ACS API Specification

A1.1 Data Structures

TYPEDEF ACS_ADDRESS

Description	ACS_ADDRESS designates a memory location for read and write functions.		
Structure	Int	Pe	Processing Element ID number
	Int	Mem	Memory location of PE
	Int	Offset	Used to allow indirect addressing

TYPEDEF ACS_CHANNEL

Description	ACS_CHANNEL is a structure that allows the user to define a channel. An array of these structures is passwd to *ACS_System_Create()*.		
Structure	Int	Src_node	Logical node number of data source
	Int	Src_port	Logical port number of data source
	Int	Des_node	Logical node number of destination
	Int	Des_port	Logical port number of destination
	Int	Window_size	Number of channel window used in flow control
	Int	Dequeue_size	Size of data dequeued from FIFO
	Int	Number	Serial number
	Void *	Dev	Reserved

TYPEDEF ACS_NODE_TYPE

Description	ACS_NODE_TYPE defines the different type of node	
enum	E_WF4	Represents Wildforce 4 board
	E_SLAAC1	Represents SLAAC-1 board
	E_SLAAC2	Represents SLAAC-2 board
	E_RCM	Represents RCM board

TYPEDEF ACS_NODE

Description	ACS_NODE is a structure that allows the user to define a type of node to be allocated by the system. An array of these structures is passwd to *ACS_System_Create()*.		
structure	Int	Number	Serial number
	ACS_NODE_TYPE	Node_type	Type of node defined in the enum ACS_NODE_TYPE.
	Int	site	Logical network address
	Void *	Dev	reserved

TYPEDEF ACS_CLOCK

Description	ACS_CLOCK is a structure that allows the user to define the properties of a clock synthesizer on an ACS board.		
structure	Double	Frequency	Synthesizer clock frequency
	Int	Countdown	Countdown timer value
	Void *	Dev	reserved

TYPEDEF ACS_CONFIG

Description	ACS_CONFIG is a structure that contains information related to a configuration.		
structure	Int	Serial_no	Configuration file ID. System uses this ID to check files in cache
	Int	Mask	Indicates which fields to set or get
	Char [255]	Label	Configuration "name" string
	Void *	Bitstream	Configuration bit data
	Int	Count	Number of bytes in configuration data
	Int	Pe_mask	Destination address or FPGA number
	Int	Cache_slot	This field is used for explicit control of cache. Configuration file will go into designate cache slot

TYPEDEF ACS_STATUS

Description	ACS_STATUS allows for a more detailed error information to be returned.		
structure	Int	Code	Error code value
	Int	Severity	Severity
	Char [255]	Text	Text string for printing

TYPEDEF ACS_WORLD_INFO

Description	ACS_WORLD_INFO is a structure that contains global and runtime information for the ACS system.		
structure	Int	Site_number	Number of workstations
	Int	Board_available	Number of boards in the system
	Int	Pe_available	Number of PEs in each board

A1.2 Support Functions

int ACS_Version(char * ver);

Description:	Get current version of the API.			
Parameters:				
	OUT	char *	ver	
Return Value:	ACS_SUCCESS			

int ACS_Initialize_Log(char * log_path);

Description:	Set network path for log; All machines in LAN should be able to access that path			
Parameters:	IN	Char *	Log_path	
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_World_Info(ACS_WORLD_INFO * world_info);

Description:	Get global data information; See ACS_WORLD_INFO Data structure			
Parameters:	IN			
	OUT	ACS_WORLD_INFO*	World_info	
Return Value:	ACS_SUCCESS			

A1.3 System Setup Functions

int ACS_Initialize(int *, char *, ACS_STATUS *);**

Description:	ACS_Initialize is responsible for parsing command line arguments, interpreting system environment variables, initializing global resources, and creating the ACS_World object; this function should be called exactly once prior to any other ACS function call.			
Parameters:	IN	Int *	Argc	command line parameters
	IN	Char ***	Argv	command line parameters
	OUT	ACS_STATUS *	Status	Get error status
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Finalize();

Description:	ACS_Finalize is the mirror of ACS_Initialize; it guarantees that allocated resources are properly freed. This includes both the system memory on the host, and allocated nodes on the network. This function should be called exactly once and the last ACS function called before exit.			
Parameters:	IN			
	OUT			
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_System_Create(ACS_SYSTEM **, ACS_NODE *, int, ACS_CHANNEL *, int, ACS_STATUS *);

Description:	ACS_System_Create is responsible for allocating the nodes specified in the node list array and creating the channels specified in the channel list array. An ACS_SYSTEM object is created and the pointer is returned in the system handle. All processors must call this function collectively to allow for appropriate information sharing.			
Parameters:	IN	ACS_SYSTEM **	System	Handle to system object pointer
	IN	ACS_NODE []	Nodes	List of nodes to allocate
	IN	Int	Node_count	Number of nodes
	IN	ACS_CHANNEL[]	Channels	List of channels to allocate
	IN	Int	channel_count	Number of channels
	OUT	ACS_STATUS *	Status	Command status
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_System_Destroy(ACS_SYSTEM * system, ACS_STATUS * status);

Description:	ACS_System_Destroy releases all nodes allocated by the system and frees all memory associated with the system object.			
Parameters:	IN	ACS_SYSTEM *	System	System to destroy
	OUT	ACS_STATUS *	Status	Command status
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Channel_Add(ACS_CHANNEL * channel, ACS_SYSTEM * system);

Description:	This function is dynamically adding a channel into the system. It appends the specified channel to the channel list in the system object.			
Parameters:	IN	ACS_CHANNEL *	Channel	Structure describing the starting and ending points and attributes of the channel to be created.
	IN	ACS_SYSTEM *	system	The system where to create this channel.
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Channel_Remove(int channel_num, ACS_SYSTEM * system);

Description:	This function removes a channel from a system.			
Parameters:	IN	Int	Channel_num	The logical channel number of the channel to be removed.
	IN	ACS_SYSTEM *	System	The system from which to remove this channel.
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Node_Add(ACS_NODE * node, ACS_SYSTEM * system);

Description:	This function allocates a node and appends it to the system object dynamically.			
Parameters:	IN	ACS_NODE *	Node	Structure describing the attributes of the node to be created
	IN	ACS_SYSTEM *	System	The system in which to create this node
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Is_Local(int node_num, ACS_SYSTEM *);

Description:	Returns ACS_ SUCCESS if node described by node_num is in the host machine (local) else ACS_FAILURE is returned.			
Parameters:	IN	Int	Node_num	The logical node number
	IN	ACS_SYSTEM *	System	The system from which to query this node
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Is_Remote(int node_num, ACS_SYSTEM *);

Description:	Returns ACS_ SUCCESS if node described by nod e_num is remote else ACS_FAILURE is returned.			
Parameters:	IN	Int	Node_num	The logical node number
	IN	ACS_SYSTEM *	System	The system from which to query this node
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.4 Memory Access Functions

int ACS_Read(void * buffer, int count, int node, ACS_ADDRESS * address, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function reads count bytes from the given address of the specified node of the system and places the data in the buffer.			
Parameters:	OUT	Void *	Buffer	Destination buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Address	Physical address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Write(void * buffer, int count, int node, ACS_ADDRESS * address, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function writes count bytes from the given buffer at the address of the specified system node.			
Parameters:	IN	Void *	Buffer	source buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Address	Physical address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.5 Board Management Functions

int ACS_Configure(ACS_CONFIG * config, int node, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function downloads a configuration bit-stream to the FPGAs of a node			
Parameters:	IN	ACS_CONFIG *	Config	Configuration data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Readback(ACS_CONFIG * config, int node, ACS_SYSTEM * system,
ACS_STATUS * status);**

Description:	This function read back a configuration file from a FPGA board.			
Parameters:	OUT	ACS_CONFIG *	Config	Buffer for readback data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Clock_Set(ACS_CLOCK * clock, int node, ACS_SYSTEM * system,
ACS_STATUS * status);**

Description:	This function sets the clock attributes of a device.			
Parameters:	IN	ACS_CLOCK *	Clock	Clock data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Clock_Get(ACS_CLOCK * clock, int node, ACS_SYSTEM * system,
ACS_STATUS * status);**

Description:	This function gets the clock attributes of a device.			
Parameters:	out	ACS_CLOCK *	Clock	Clock data
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Run(int node, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function starts the board clock.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Stop(int node, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function stops the board clock.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Reset(int node, ACS_SYSTEM * system, int pe_mask, int enable,
ACS_STATUS * status);**

Description:	This function either enables or disables a RESET signal on a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	IN	Int	Pe_mask	Mask used to designate processing elements affected by reset.
	IN	Int	Enable	If nonzero, reset signal is enabled, otherwise disabled
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Reset_Toggle(int node, ACS_SYSTEM * system, int pe_mask,
ACS_STATUS * status);**

Description:	Sends a reset pulse to a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	IN	Int	Pe_mask	Mask used to designate processing elements affected by reset.
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Interrupt(int node, int pe, int number, ACS_SYSTEM * system,
ACS_STATUS * status);**

Description:	This function causes an interrupt signal of the specified number to be sent to a node device. This function has no effect on WildForce nodes.			
Parameters:	IN	Int	Node	Logical node number
	IN	Int	Pe_mask	
	IN	Int	Number	Interrupt number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_InterruptPoll(int node, int pe, int * result_mask, int number,
ACS_SYSTEM * system, ACS_STATUS * status);**

Description:	Checks the interrupt status of specified processing elements on a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	Int	Pe_mask	Mask used to designate processing elements affected by reset.
	OUT	Int *	Result_mask	Return a 0 when all processing elements designated by pe_mask have raised interrupt. Otherwise, gives mask of PEs that have not yet raised interrupts
	IN	Int	Number	Interrupt number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_InterruptAck(int node, int pe, int number, ACS_SYSTEM * system,
ACS_STATUS * status);**

Description:	Acknowledges interrupts of specified processing elements on a node device.			
Parameters:	IN	Int	Node	Logical node number
	IN	Int	Pe_mask	Mask used to designate PEs ack' d
	IN	Int	Number	Interrupt number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Reg_Read(ACS_SYSTEM * system, int node_num, int reg_id,
ACS_REGISTER *);**

Description:	Read data from node register			
Parameters:	IN	ACS_SYSTEM *	System	System object
	IN	Int	Node	Logical node number
	IN	Int	Reg_id	Register id number
	IN	ACS_REGISTER *	Reg	Register data structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Reg_Write(ACS_SYSTEM * system, int node_num, int reg_id,
ACS_REGISTER *);**

Description:	Write date into node's register			
Parameters:	IN	ACS_SYSTEM *	System	System object
	IN	Int	Node	Logical node number
	IN	Int	Reg_id	Register id number
	IN	ACS_REGISTER *	Reg	Register data structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.6 Streaming Data Functions

**int ACS_Enqueue(void * src, int size, int port, ACS_SYSTEM *,
ACS_STATUS *);**

Description:	This function puts data into a specific host port number.			
Parameters:	IN	Void *	Src	Source buffer for data
	IN	Int	Size	Number of bytes to write
	IN	Int	Port	System port number to write
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

**int ACS_Dequeue(void * des, int size, int port, ACS_SYSTEM *,
ACS_STATUS *);**

Description:	This function takes data from a specific host port number			
Parameters:	IN	Void *	des	Destination buffer for data
	IN	Int	Size	Number of bytes to read
	IN	Int	Port	System port number to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.7 Group Management

**int ACS_Group_Create(ACS_GROUP ** group, int nodes[], int count,
ACS_SYSTEM * system);**

Description:	ACS_Group_Create is the ACS_GROUP object constructor. The nodes argument is an array of logical node numbers (integers) of length count. The array index defines the new logical node number in the new group context. Group creation can be nested (system can be an object of type ACS_GROUP).			
Parameters:	OUT	ACS_GROUP **	Group	Handle to return created group object.
	IN	Int []	Nodes	Array of logical node numbers
	IN	Int	Count	Length of the nodes array
	IN	ACS_SYSTEM *	System	System object
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Group_Destroy(ACS_GROUP * group);

Description:	ACS_Group_Destroy is the ACS_GROUP object destructor. Unlike ACS_System_Destroy, this function does not free allocated nodes or other resources associated with the system.			
Parameters:	IN	ACS_GROUP *	Group	Group object to be free
	OUT			
Return Value:	ACS_SUCCESS / ACS_FAILURE			

A1.8 Non-blocking function

int ACS_Request_Create(ACS_REQUEST ** request, int number);

Description:	ACS_Request_Create is the ACS_REQUEST object constructor. The number argument specifies the number of commands the request object can store.			
Parameters:	IN	ACS_REQUEST **	request	Handle to request object pointer
	IN	Int	Number	Number of command entries to allocate
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Request_Destroy(ACS_REQUEST * request);

Description:	ACS_Request_Create is the ACS_REQUEST objec t destructor. It frees the memory associated with a request object.			
Parameters:	IN	ACS_REQUEST *	Request	Object to destroy
	OUT			
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Commit(ACS_REQUEST * request, ACS_REQUEST_STAT * status);

Descript ion:	ACS_Commit processes the list of commands in the request structure, and issues the commands to the nodes. After ACS_Commit returns, any output buffers may be reused (such as from an ACS_Write or ACS_Enqueue command). However, the input buffers (such as from an ACS_Read) are undefined.			
Parameters:	IN	ACS_REQUEST *	Request	Request to submit
	IN	ACS_SYSTEM *	System	System group on which to commit request
	OUT	ACS_REQUEST_STAT *	Status	Command status array
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_Wait(ACS_REQUEST * request, ACS_REQUEST_STAT * status);

Description:	This function causes the thread to block until all commands have completed.			
Parameters:	IN	ACS_REQUEST *	Request	Request to submit
	IN	ACS_SYSTEM *	System	System group on wh ich to commit request
	OUT	ACS_REQUEST_STAT *	Status	Command status array
Return Value:	ACS_SUCCESS – all commands in the request have been successfully completed. ACS_FAILURE – the request has been completed. However, some of the commands have returned an error condition. The programmer should check the status structure to determine which commands have failed.			

int ACS_Test(ACS_REQUEST * request, ACS_REQUEST_STAT * status);

Description:	This function tests the request object to determine if all commands have been completed. Compared with ACS_Wait, ACS_Test only tests, but does not wait until the request has been finished. It's a non-blocking call.			
Parameters:	IN	ACS_REQUEST *	Request	Request to submit
	IN	ACS_SYSTEM *	System	System group on which to commit request
	OUT	ACS_REQUEST_STAT *	Status	Command status array
Return Value:	ACS_SUCCESS – all commands in the request have been successfully completed. ACS_FAILURE – the request has been completed. However some of the commands have returned an error condition. The programmer should check the status structure to determine which commands have failed.			

int ACS_ReadN(ACS_REQUEST * request, void * buffer, int count, int node, ACS_ADDRESS * addr, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function reads count bytes from the given address of the specified node in the system and places the data in the buffer.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	OUT	Void *	Buffer	Destination buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Addr	Address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_WriteN(ACS_REQUEST * request, void * buffer, int count, int node, ACS_ADDRESS * addr, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function writes count bytes from the given buffer at the address of the specified system node.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	OUT	Void *	Buffer	Source buffer for data
	IN	Int	Count	Number of bytes to read
	IN	Int	Node	Logical node number
	IN	ACS_ADDRESS *	Addr	Address at node to read
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_ResetN(ACS_REQUEST * request, int node, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function causes a RESET signal to be sent to a node device.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	IN	Int	Node	Logical node number
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

int ACS_ConfigureN(ACS_REQUEST * request, int node, ACS_CONFIG * config, ACS_SYSTEM * system, ACS_STATUS * status);

Description:	This function send configuration bit stream to a node device.			
Parameters:	OUT	ACS_REQUEST *	Request	Request to append to
	IN	Int	Node	Logical node number
	IN	ACS_CONFIG *	Config	Configuration file
	IN	ACS_SYSTEM *	System	System object
	OUT	ACS_STATUS *	Status	Command status structure
Return Value:	ACS_SUCCESS / ACS_FAILURE			

Appendix 2. Testing code for ACS API

A2.1 Memory Access

```
/** This program is to test the memory access functions, ACS_Read() and ACS_Write(). The program tests
the memory access function in both local and remote machines. The program calls ACS_Write() to write
various sizes of data into the board, local and remote. It then reads data back from the boards through
ACS_Read(). After that, the program compares the writing data and the reading data. */

#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"
#include "mpi.h"

#define MEMORY_SIZE            1024
#define PASS_THROUGH                    10

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);

void main(int argc, char ** argv)
{
        ACS_STATUS    status;
        char          version[0x10];

        // Get the SLAAC API version
        ACS_Version(version);
        cout<<"SLAAC APIs version "<<version<<endl;

        // Initialize the SLAAC system.
        cout<<"ACS Initializing...";
        ACS_Initialize(&argc, &argv, &status);
        cout<<"done"<<endl;

        ACS_WORLD_INFO wf;
        ACS_World_Info(&wf);
        int sites = wf.site_number;

        // Construct the network topology
        ACS_CHANNEL                   channels[0x10];
        ACS_NODE              nodes[0x10];
        Node_initialize(nodes, sites);

        // Creating the ACS system...
        cout<<"Creating system...";
        ACS_SYSTEM     * system;
        ACS_System_ Create(&system, nodes, sites, channels, 0, &status);
        cout<<"done. node number: "<<sites<<endl<<endl;

        cout<<endl<<endl<<endl<<"Now testing Memory function..."<<endl;
```

```
	char in_buffer[MEMORY_SIZE];
	char out_buffer[MEMORY_SIZE];

	// initialize buffer
	for (int i=0; i<MEMORY_SIZE; i++)
		out_buffer[i] = (i % 256);

	ACS_ADDRESS address;
	address.mem			= 0;
	address.offset	= 0;

	int success;

	double start_t, end_t;

	for (i=0; i<sites; i++) {
		cout<<"Testing node "<<i<<"..."<<endl;
		for (int k=0; k<3; k++) {
			address.pe = WF4_PE(k);
			for (int j=4; j<MEMORY_SIZE; j+=128) {
				ACS_Write (out_buffer, j, i, &address, system, &status);
				start_t = MPI_Wtime();
				ACS_Read (in_buffer, j, i, &address, system, &status);
				end_t = MPI_Wtime();
				cout<<"Reading PE("<<k<<") in "<<j<<" bytes
			in"<<"\t"<<(end_t-start_t)<<" seconds"<<endl;

				if (memcmp(in_buffer, out_buffer, j) == 0) {
				}
				else {
					cout<<"ERROR!"<<endl;
					success = FALSE;
				}
				memset(in_buffer, 0, j);
			}
		}
	}

	// Destroy the acs_system, release the resource.
	cout<<"Destorying system...";
	ACS_System_Destroy(system, &status);
	cout<<"done"<<endl;

	// finalize the acs world, destroy throughly.
	cout<<"ACS_Finalize...";
	ACS_Finalize();
	cout<<"done."<<endl<<endl;

	cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
	for (int i=0; i<sites; i++) {
		memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
		nodes[i].site = i;
```

A2.2 Streaming Data

A2.2.a Measuring RTT Value

```
/** This program is to test the ACS_Enqueue() and ACS_Dequeue() as well as the board management
functions, including ACS_Configure(), ACS_Reset() and ACS_Run(). The program first links variety
number of nodes into a ring, then uses board management functions to configure the board, start the clock,
and send the RESET signal to the board. It then calls ACS_Enqueue() to put data into the system object.
The data should pass through each board in the ring automatically and finally go back to the system object.
The test program then calls ACS_Dequeue() to get the data back from the system object.

This program measures one fact of the performance for the data stream functions, the round trip time
(RTT). The RTT is the point of total time for putting data into the system, passing through and processing
data in each board, and getting data back from the system. The RTT is measured by enqueues a range of
size of data into the system */


#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"

#define FIFO_SIZE                        1024
#define PASS_THROUGH                     1000

#define WF4_CLOCK_FREQUENCY 24.576

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM *, int, int);
DWORD *readBitFile (const char *filename, int *numBytes);
double    pt_time[PASS_THROUGH];

void main(int argc, char ** argv)
{
        ACS_STATUS    status;
        char          version[0x10];

        // Initialize the SLAAC system.
        cout<<"ACS Initializing...";
        ACS_Initialize(&argc, &argv, &status);
        cout<<"done"<<endl;

        // Get the ACS world information.
        // How many workstations are in the system.
        ACS_WORLD_INFO wf;
        ACS_World_Info(&wf);
        int sites = wf.site_number;

        // Construct the network topology
        ACS_CHANNEL                  channels[0x10];
```

```
ACS_NODE                nodes[0x10];
Node_initialize(nodes, sites);
Channel_initialize(channels, sites);

// Creating the ACS system...
cout<<"Creating system...";
ACS_SYSTEM     * system;
ACS_System_Create(&system, nodes, s ites, channels, sites + 1, &status);
cout<<"done. node number: "<<sites<<", linked by a ring"<<endl<<endl;

char        in_buffer[FIFO_SIZE];
char        out_buffer[FIFO_SIZE];
int                 success = TRUE;

for (int j=0; j<FIFO_SIZE; j++)
            out_buffer[j] = (j % 256);

for (int i=0; i<sites; i++)
            Config_Nodes(system, nodes[i].number, i);

for (i=0; i<sites; i++)
            ACS_Run(nodes[i].number, system, &status);

int dequeue;
for (i=0; i<PASS_THROUGH; i++) {
            memset(in_buffer, 0, FIFO_SIZE);

            double start_t = MPI_Wtime();
            ACS_Enqueue(out_buffer, FIFO_SIZE, 0, system, &status);

            dequeue = 0;
            while (dequeue != FIFO_SIZE)
                        dequeue += ACS_Dequeue(in_buffer, FIFO_SIZE  -
                        dequeue, 0, system, &status);
            double end_t = MPI_Wtime();
            pt_time[i] = (end_t - start_t);
}

double total_time = 0;
int invalid = 0;
for (i=1; i<PASS_THROUGH; i++) {
            if ((int)(pt_time[i] / pt_time[i-1]) > 2) {
                        pt_time[i] = pt_time[i-1];
                        invalid ++;
            }
            else {
                        total_time += pt_time[i];
            }
}

cout<<"The average RTT values is  "<<(total_time / (PASS_THROUGH -
invalid))<<" seconds"<<endl;

cout<<"total invalid numer is "<<invalid<<endl;

// Destroy the acs_system, release the resource.
Sleep(3000);
```

```
        cout<<"Destorying system...";
        ACS_System_Destroy(system, &status);
        cout<<"done"<<endl;

        // finalize the acs world, destroy throughly.
        cout<<"ACS_Finalize...";
        ACS_Finalize();
        cout<<"done."<<endl<<endl;

        cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}



void Node_initialize(ACS_NODE nodes[], int sites)
{
        for (int i=0; i<sites; i++) {
                memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
                nodes[i].site = i;
        }
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
        for (int i=1; i<sites; i++) {
                channels[i].src_node        = i-1;
                channels[i].src_port        = WF4_Fifo_Pe4;
                channels[i].des_node        = i;
                channels[i].des_port        = WF4_Fifo_Pe1;
                channels[i].window_size = 10;
                channels[i].dequeue_size = 1024;
                channels[i].number          = 0;
        }
        channels[0].src_node                = ACS_HOST_NUM(0);
        channels[0].src_port                = 0;
        channels[0].des_node                = 0;
        channels[0].des_port                = WF4_Fifo_Pe1;
        channels[0].window_size             = 10;
        channels[0].dequeue_size  = 1024;
        channels[0].number                  = 0;

        channels[sites].src_node   = sites - 1;
        channels[sites].src_port   = WF4_Fifo_Pe4;
        channels[sites].des_node   = ACS_HOST_NUM(0);
        channels[sites].des_port   = 0;
        channels[sites].window_size         = 10;
        channels[sites].dequeue_size        = 1024;
        channels[sites].number              = 0;
}

void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id)
{
        int             i;
        ACS_CONFIG      config;
        ACS_CLOCK       clock;
        char            configFileName[256];
        ACS_STATUS      status;
```

```
	char		boardname[256];
	int		serial_no = 0;


	sprintf(config.label, "Passthrough Test");
	sprintf(boardname, "board%d", board_id);

	for (i = 0; i < WF4_MAX_PES; i++) {
		sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?"":"c", i);
		printf("config File %s\n",configFileName);
		config.bitstream = readBitFile (configFileName, &(config.count));
		config.serial_no = serial_no++;

		if (config.bitstream == NULL) {
			printf ("initSlaac: NULL configuration buffer!  Exiting.\n");
			exit (-1);
		}

		config.pe_mask = WF4_PE (i);

		ACS_Configure(&config, node_id, system, &status);
		delete [] config.bitstream;
	}

	// Start the clock
	clock.frequency = WF4_CLOCK_FREQUENCY;
	clock.countdown = 0;
	if (ACS_Clock_Set(&clock, node_id, system, &status) != ACS_SUCCESS)
		printf ("initSlaac: ClockSet failed.\n");

	if (ACS_Reset(node_id, system, ACS_PE1, TRUE, &status) != ACS_SUCCESS)
		printf ("initSlaac: Failed to send reset.\n");

	printf ("Clock set to %.3f MHz.\n", WF4_CLOCK_FREQUENCY);
}

DWORD *readBitFile (const char *filename, int *numBytes)
{
	FILE *inFile;
	unsigned char *buffer;
	long fileSize;

	// try to open the input file
	if ((inFile = fopen (filename, "rb")) == NULL) {
		*numBytes = 0;
		printf("readBitFile: Couldn't open file '%s\n", filename);
		return NULL;
	}

	// figure out how long the file is
	fseek (inFile, 0, SEEK_END);
	fileSize = ftell (inFile);
	*numBytes = (int) fileSize;
	// make sure numBytes is accurate...
	if (*numBytes != fileSize) {
		printf ("readBitFile: Oh, no!  File '%s' is bigger than an
	int!\n", filename);
```

```
                fclose (inFile);
                return NULL;
        }

        // make the new buffer
        buffer = new unsigned char [fileSize];

        // read in the bytes
        fseek (inFile, 0, SEEK_SET);
        fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);
        return (DWORD *) buffer;
}
```

A2.2.b Measuring throughput

```
/** This program differs from the RTT test program by creating a thread to dequeue the data instead of
calling ACS_Dequeue() in the host program. The host program then keeps enqueuing five mega bytes data
into the system and records the time of enqueuing every 1024 bytes data. */

#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"
#include "mpi.h"

#define FIFO_SIZE                   1024
#define PASS_THROUGH                2500

#define WF4_CLOCK_FREQUENCY 24.576

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM * , int, int);
DWORD *readBitFile (const char *filename, int *numBytes);
void Dequeue_thread(void *);

char    in_buffer[FIFO_SIZE];
char    out_buffer[FIFO_SIZE];

ACS_SYSTEM    *      acs_system;
ACS_CHANNEL                  channels[0x10];
ACS_NODE             nodes[0x10];

int thread_quit = 0;
int quit = 0;

double pt_time[PASS_THROUGH];

void main(int argc, char ** argv)
{
        ACS_STATUS    status;
        char          version[0x10];

        // Initialize the SLAAC system.
        cout<<"ACS Initializing...";
        ACS_Initialize(&argc, &argv, &status);
        cout<<"done"<<endl;

        // Initialize log file. make sure the path is a network path so
        // each node can visit it
        cout<<"Log Initializing...";
        ACS_Initialize_Log("\\\\tuba\\home\\mpi\\");
        cout<<"done"<<endl;

        // Get the ACS world information.
        // How many nodes participated in the computation.
```

```
	ACS_WORLD_INFO wf;
	ACS_World_Info(&wf);
	int sites = wf.site_number;

	Node_initialize(nodes, sites);
	Channel_initialize(channels, sites);

	// Creating the ACS system...
	cout<<"Creating system...";
	ACS_System_Create(&acs_system, nod es, sites, channels, sites + 1, &status);
	cout<<"done. node number: "<<sites<<", linked by a ring"<<endl<<endl;

	int		success = TRUE;

	for (int j=0; j<FIFO_SIZE; j++)
		out_buffer[j] = (j % 256);

	for (int i=0; i<sites; i++)
		Config_Nodes(acs_system,  nodes[i].number, i);

	printf("Sleeping....");
	for (i=0; i<4; i++) {
		ACS_SLEEP(1000);
		printf(".");
	}
	printf("\nWake up\n");

	ACS_THREAD_SPAWN(Dequeue_thread);

	for (i=0; i<PASS_THROUGH; i++) {
		double start_t = MPI_Wtime();
		ACS_Enqueue(out_buffer, FIFO_SIZE, 0, acs_system, &status);
		double end_t = MPI_Wtime();
		pt_time[i] = end_t - start_t;
		printf("%d\n", i);
	}

#define _SKIP 500

double total_time = 0;
	int invalid = 0;

	long total_data = (long)FIFO_SIZE * (PASS_THROUGH - invalid - _SKIP);
	double throughput = (double)total_data / total_time;

	cout<<"The average throughput is "<<throughput<<" bytes/seconds"<<endl;
	cout<<"total invilid number "<<invalid<<endl;

	quit = 1;
	printf("wait until thread quit");
	while (!thread_quit);

	cout<<endl<<"Sleeping....";
	for (i=0; i<4; i++) {
		ACS_SLEEP(1000);
		printf(".");
	}
```

```
		printf("\nWake up\n");

		cout<<"Destorying system...";
		ACS_System_Destroy(acs_system, &status);
		cout<<"done"<<endl;

		// finalize the acs world, destroy throughly.
		cout<<"ACS_Finalize...";
		ACS_Finalize();
		cout<<"done."<<endl<<endl;

		cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
		for (int i=0; i<sites; i++) {
				memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
				nodes[i].site = i;
		}
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
		for (int i=1; i<sites; i++) {
				channels[i].src_node		= i-1;
				channels[i].src_port		= WF4_Fifo_Pe4;
				channels[i].des_node		= i;
				channels[i].des_port		= WF4_Fifo_Pe1;
				channels[i].window_size = 10;
				channels[i].dequeue_size= 1024;
				channels[i].number		= 0;
		}
		channels[0].src_node			= ACS_HOST_NUM(0);
		channels[0].src_port			= 0;
		channels[0].des_node			= 0;
		channels[0].des_port			= WF4_Fifo_Pe1;
		channels[0].window_size			= 10;
		channels[0].dequeue_size  = 1024;
		channels[0].number			= 0;

		channels[sites].src_node   = sites - 1;
		channels[sites].src_port   = WF4_Fifo_Pe4;
		channels[sites].des_node   = ACS_HOST_NUM(0);
		channels[sites].des_port   = 0;
		channels[sites].window_size		= 10;
		channels[sites].dequeue_size		= 1024;
		channels[sites].number			= 0;
}

void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id)
{
		int				i;
		ACS_CONFIG	config;
		ACS_CLOCK	clock;
		char		configFileName[256];
		ACS_STATUS	status;
```

```
	char			boardname[256];
	int			serial_no = 0;


	sprintf(config.label, "Passthrough Test");
	sprintf(boardname, "board%d", board_id);

	for (i = 0; i < WF4_MAX_PES; i++) {
		sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?"":"c", i);
		printf("config File %s\n",configFileName);
		config.bitstream = readBitFile (configFileName, &(config.count));
		config.serial_no = serial_no++;

		if (config.bitstream == NULL) {
			printf ("initSlaac: NULL configuration buffer!  Exiting.\n");
			exit (-1);
		}
		config.pe_mask = WF4_PE (i);
		ACS_Configure(&config, node_id, system, &status);
		delete [] config.bitstream;
	}

	// Start the clock
	clock.frequency = WF4_CLOCK_FREQUENCY;
	clock.countdown = 0;
	if (ACS_Clock_Set(&clock, node_id, system, &status) != ACS_SUCCESS)
		printf("initSlaac: ClockSet failed.\n");

	// Run the board
	if (ACS_Run(node_id, system, &status) != ACS_SUCCESS)
		printf ("initSlaac: Failed to start clock.\n");

	// Reset the board
	if (ACS_Reset(node_id, system, ACS_PE1, TRUE, &status) != ACS_SUCCESS)
		printf ("initSlaac: Failed to send reset.\n");

	printf ("Clock set to %.3f MHz.\n", WF4_CLOCK_FREQUENCY);
}

DWORD *readBitFile (const char *filename, int *numBytes)
{
	FILE *inFile;
	unsigned char *buffer;
	long fileSize;

	// try to open the input file
	if ((inFile = fopen (filename, "rb")) == NULL) {
		*numBytes = 0;
		printf ("readBitFile: Couldn't open file '%s\n", filename);
		return NULL;
	}

	// figure out how long the file is
	fseek (inFile, 0, SEEK_END);
	fileSize = ftell (inFile);
	*numBytes = (int) fileSize;
	// make sure numBytes is accurate...
	if (*numBytes != fileSize) {
		printf ("readBitFile: Oh, no!  File '%s' is bigger than an
	int!\n", filename);
		fclose (inFile);
		return NULL;
	}

	// make the new buffer
	buffer = new unsigned char [fileSize];

	// read in the bytes
	fseek (inFile, 0, SEEK_SET);
	fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);

	return (DWORD *) buffer;
}

void Dequeue_thread(void *)
{
	long dequeue = 0;
	ACS_STATUS status;

	while (!quit) {
		dequeue += ACS_Dequeue(in_buffer, FIFO_SIZE, 0, acs_system, &status);
	}

	printf("dequeued %d bytes data\n", dequeue);

	thread_quit = 1;
}
```

A2.3 Non-blocking

```
/** This program illustrates how the user takes advantage of using non-blocking functions to reduce the
total execution time through overlapping the computation and communication */

#include <stdio.h>
#include <iostream.h>
#include "mpi.h"
#include "acs.h"
#include "CWildForce4Node.h"

#define MEMORY_SIZE    1024000

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);

char out_buffer[MEMORY_SIZE];
char in_buffer[MEMORY_SIZE];

void main(int argc, char ** argv)
{
        ACS_STATUS      status;
        char            version[0x10];

        // Get the SLAAC API version
        ACS_Version(version);
        cout<<"SLAAC APIs version "<<version<<endl;

        // Initialize the SLAAC system.
        cout<<"ACS Initializing...";
        ACS_Initialize(&argc, &argv, &status);
        cout<<"done"<<endl;

        // Get to know the ACS world information.
        // How many nodes participated in the computation.
        ACS_WORLD_INFO wf;
        ACS_World_Info(&wf);
        int sites = wf.site_number;

        // Construct the network topology
        ACS_CHANNEL                     channels[0x10];
        ACS_NODE                nodes[0x10];
        Node_initialize(nodes, sites);
        //Channel_initialize(channels, sites);

        // Creating the ACS system...
        cout<<"Creating system...";
        ACS_SYSTEM      * system;
        ACS_System_Create(&system, nodes, sites, channels, 0, &status);

        ACS_REQUEST * request;
        ACS_REQUEST_STAT r_status;
```

```
ACS_ADDRESS addr;


for (int i=0; i<MEMORY_SIZE; i++) out_buffer[i] = (i % 256);

ACS_Request_Create(&request, 5);

addr.mem            = 0;
addr.offset         = 0;
addr.pe   = WF4_PE(0);

double start_t, end_t;

#define _REPEAT             50
#define _REPEAT2            5

start_t = MPI_Wtime();
for (int j=0; j<_REPEAT2; j++) {
            ACS_Write(out_buffer, MEMORY_SIZE, nodes[1].number, &addr,
            system, &status);
            ACS_Read(in_buffer, MEMORY_SIZE, nodes[1].number, &addr,
            system, &status);
}
for (j=0; j<_REPEAT; j++) {
            ACS_Write(out_buffer, MEMORY_SIZ E, nodes[0].number, &addr,
            system, &status);
            ACS_Read(in_buffer, MEMORY_SIZE, nodes[0].number, &addr,
            system, &status);
}
end_t = MPI_Wtime();
printf("\nTotal time for blocking call is %f\n", (end_t - start_t));

start_t = MPI_Wtime();
for (j=0; j<_REPEAT2; j++) {
            ACS_WriteN(request, out_buffer, MEMORY_SIZE,
            nodes[1].number, &addr, system, &status);
            ACS_ReadN(request, in_buffer, MEMORY_SIZE, nodes[1].number,
&addr, system, &status);
}
ACS_Commit(request, &r_status);
for (j=0; j<_REPEAT; j++) {
            ACS_Write(out_buffer, MEMORY_SIZE, nodes[0].number, &addr,
            system, &status);
            ACS_Read(in_buffer, MEMORY_SIZE, nodes[0].number, &addr,
            system, &status);
}
ACS_Wait(request, &r_status);
end_t = MPI_Wtime();
printf("\nTotal time for non-blocking call is %f\n", (end_t - start_t));

ACS_Request_Destroy(request);

// Destroy the acs_system, release the resource.
cout<<"Destorying system...";
ACS_System_Destroy(system, &status);
cout<<"done"<<endl;
```

```
        // finalize the acs world, destroy throughly.
        cout<<"ACS_Finalize...";
        ACS_Finalize();
        cout<<"done."<<endl<<endl;

        cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
        for (int i=0; i<sites; i++) {
                memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
                nodes[i].site = i;
        }
}
```

A2.4 Testing ACS_ALL

```
/** This program is revised from the data stream test program. Instead of using an individual node id in
calling the board management functions, it uses ACS_ALL. So the user only needs to call those functions
once. */

#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "CWildForce4Node.h"

#define WF4_CLOCK_FREQUENCY 24.576

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id);
DWORD *readBitFile (const char *filename, int *numBytes);

void main(int argc, char ** argv)
{
        ACS_STATUS    status;
        char          version[0x10];

        // Get the SLAAC API version
        ACS_Version(version);
        cout<<"SLAAC APIs version "<<version<<endl;

        // Initialize the SLAAC system.
        cout<<"ACS Initializing...";
        ACS_Initialize(&argc, &argv, &status);
        cout<<"done"<<endl;

        // Get the ACS world information.
        // How many nodes participated in the computation.
        ACS_WORLD_INFO wf;
        ACS_World_Info(&wf);
        int sites = wf.site_number;

        // Construct the network topology
        ACS_CHANNEL                channels[0x10];
        ACS_NODE            nodes[0x10];
        Node_initialize(nodes, sites);
        Channel_initialize(channels, sites);

        // Creating the ACS system...
        cout<<"Creating system...";
        ACS_SYSTEM     * system;
        ACS_System_Create(&system, nodes, sites, channels, sites + 1, &status);
        cout<<"done. node number: "<<sites<<", linked by a ring"<<endl<<endl;

        for (int i=0; i<sites; i++)
                Config_Nodes(system, nodes[i].number, i);
        ACS_CLOCK       clock;
        clock.frequency = WF4_CLOCK_FREQUENCY;
        clock.countdown = 0;

        if (ACS_Clock_Set(&clock, ACS_ALL, system, &status) != ACS_SUCCESS)
                printf ("initSlaac: ClockSet failed.\n");

        if (ACS_Run(ACS_ALL, system, &status) != ACS_SUCCESS)
                printf ("initSlaac: Failed to start clock.\n");
        printf ("before reset\n");

        if (ACS_Reset(ACS_ALL, system, ACS_PE1, TRUE, &status) != ACS_SUCCESS)
                printf ("initSlaac: Failedto send reset.\n");

        printf ("Clock set to %.3f MHz.\n", WF4_CLOCK_FREQUENCY);


        // Destroy the acs_system, release the resource.
        cout<<"Destorying system...";
        ACS_System_Destroy(system, &status);
        cout<<"done"<<endl;

        // finalize the acs world, destroy throughly.
        cout<<"ACS_Finalize...";
        ACS_Finalize();
        cout<<"done."<<endl<<endl;

        cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
        for (int i=0; i<sites; i++) {
                memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
                nodes[i].site = i;
        }
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
        for (int i=1; i<sites; i++) {
                channels[i].src_node = i-1;
                channels[i].src_port = WF4_Fifo_Pe4;
                channels[i].des_node = i;
                channels[i].des_port = WF4_Fifo_Pe1;
        }
        channels[0].src_node        = ACS_HOST_NUM(0);
        channels[0].src_port        = 0;
        channels[0].des_node        = 0;
        channels[0].des_port        = WF4_Fifo_Pe1;

        channels[sites].src_node = sites - 1;
        channels[sites].src_port = WF4_Fifo_Pe4;
        channels[sites].des_node = ACS_HOST_NUM(0);
        channels[sites].des_port = 0;
}
```

```
void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id)
{
	int				i;
	ACS_CONFIG	config;
	char		configFileName[256];
	ACS_STATUS	status;
	char		boardname[256];

	int				serial_no = 0;


	sprintf(config.label, "Passthrough Test");
	sprintf(boardname, "board%d", board_id);

	for (i = 0; i < WF4_MAX_PES; i++) {
		sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?"":"c", i);
		printf("config File %s\n",configFileName);
		config.bitstream = readBitFile (configFileName, &(config.count));
		config.serial_no = serial_no++;

		if (config.bitstream == NULL) {
			printf ("initSlaac: NULL configuration buffer!  Exiting.\n");
			exit (-1);
		}

		config.pe_mask = WF4_PE (i);

		ACS_Configure(&config, node_id, system, &status);
		delete [] config.bitstream;
	}
}

DWORD *readBitFile (const char *filename, int *numBytes)
{
	FILE *inFile;
	unsigned char *buffer;
	long fileSize;

	// try to open the input file
	if ((inFile = fopen (filename, "rb")) == NULL) {
		*numBytes = 0;
		printf ("readBitFile: Couldn't open file '%s\n", filename);
		return NULL;
	}

	// figure out how long the file is
	fseek (inFile, 0, SEEK_END);
	fileSize = ftell (inFile);
	*numBytes = (int) fileSize;
	// make sure numBytes is accurate...
	if (*numBytes != fileSize) {
		printf ("readBitFile: Oh, no!  File '%s' is bigger than an
	int!\n", filename);
		fclose (inFile);
		return NULL;
	}
```

```
        // make the new buffer
        buffer = new unsigned char [fileSize];

        // read in the bytes
        fseek (inFile, 0, SEEK_SET);
        fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);

        return (DWORD *) buffer;
}
```

A2.5 Cache configuration file

```
/** In this test, the host program sends three different configuration files to the remote node repeatedly.
Since the configuration files have been cached in the remote machine during the first time when it was
configured, the system will not send any actual data to the remote machine in the later time. The time of
reconfiguring the board has been significantly reduced after the first time. */

#include <stdio.h>
#include <iostream.h>
#include "acs.h"
#include "acs_system.h"
#include "CWildForce4Node.h"

_SYSTEM_TYPE(CWildForce4Node);

void Node_initialize(ACS_NODE nodes[], int sites);
void Channel_initialize(ACS_CHANNEL channels[], int sites);
void Config_Nodes(ACS_SYSTEM * system, int node_id, int board_id);
DWORD *readBitFile (const char *filename, int *numBytes) ;

void main(int argc, char ** argv)
{
        ACS_STATUS      status;
        char            version[0x10];

        // Get the SLAAC API version
        ACS_Version(version);
        cout<<"SLAAC APIs version "<<version<<endl;

        // Initialize the SLAAC system.
        cout<<"ACS Initializing...";
        ACS_Initialize(&argc, &argv, &status);
        cout<<"done"<<endl;

        // Initialize log file. make sure the path is a network path so
        // each node can visit it
        cout<<"Log Initializing...";
        ACS_Initialize_Log("\\\\tuba\\home\\mpi\\");
        cout<<"done"<<endl;

        // Get to know the ACS world information.
        // How many nodes participated in the computation.
        ACS_WORLD_INFO wf;
        ACS_World_Info(&wf);
        int sites = wf.site_number;

        // Construct the network topology
        ACS_CHANNEL                     channels[0x10];
        ACS_NODE                nodes[0x10];
        Node_initialize(nodes, sites);
        Channel_initialize(channels, sites);

        // Creating the ACS system...
        cout<<"Creating system...";
        ACS_SYSTEM     * system;
        ACS_System_Create(&system, nodes, sites, channels, sites + 1, &status);
        cout<<"done. node number: "<<sites<<", linked by a ring"<<endl<<endl;

        double start_t, end_t;

        for (int i=0; i<2; i++) {
                for (int j=0; j<sites; j++) {
                        start_t = MPI_Wtime();
                        Config_Nodes(system, nodes[j].number, j);
                        end_t = MPI_Wtime();
                        cout<<endl<<endl<<"Send configure to node "<<j<<" in
                        "<<(end_t-start_t)<<" seconds"<<endl;
                }
                Sleep(2000);
        }

        // Destroy the acs_system, release the resource.
        cout<<"Destorying system...";
        ACS_System_Destroy(system, &status);
        cout<<"done"<<endl;

        // finalize the acs world, destroy throughly.
        cout<<"ACS_Finalize...";
        ACS_Finalize();
        cout<<"done."<<endl<<endl;

        cout<<"VPI & SU, ECPE, CCM Lab. Fall 1999"<<endl;
}

void Node_initialize(ACS_NODE nodes[], int sites)
{
        for (int i=0; i<sites; i++) {
                memset((void *)&nodes[i], 0, sizeof(ACS_NODE));
                nodes[i].site = i;
        }
}

void Channel_initialize(ACS_CHANNEL channels[], int sites)
{
        for (int i=1; i<sites; i++) {
                channels[i].src_node = i-1;
                channels[i].src_port = WF4_Fifo_Pe4;
                channels[i].des_node = i;
                channels[i].des_port = WF4_Fifo_Pe1;
        }
        channels[0].src_node = ACS_HOST_NUM(0);
        channels[0].src_port = 0;
        channels[0].des_node = 0;
        channels[0].des_port = WF4_Fifo_Pe1;

        channels[sites].src_node = sites - 1;
        channels[sites].src_port = WF4_Fifo_Pe4;
        channels[sites].des_node = ACS_HOST_NUM(0);
        channels[sites].des_port = 0;
}
```

```
void Config_Nodes(ACS_SYSTEM * system, in t node_id, int board_id)
{
	int				i;
	ACS_CONFIG	config;
	ACS_CLOCK	clock;
	char		configFileName[256];
	ACS_STATUS	status;
	char		boardname[256];

	int				serial_no = 0;


	sprintf(config.label, "Passthrough Test");
	sprintf(boardname, "board%d", board_id);

	for (i = 0; i < WF4_MAX_PES; i++) {
		sprintf (configFileName, "%s\\%spe%d.x86", boardname, (i)?"":"c",i);
		printf("config File %s\n",configFileName);
		config.bitstream = readBitFile (configFileName, &(config.count));
		config.serial_no = serial_no++;

		if (config.bitstream == NULL) {
			printf ("initSlaac: NULL configuration buffer!  Exiting.\n");
			exit (-1);
		}
		config.pe_mask = WF4_PE (i);
		ACS_Configure(&config, node_id, system, &status);
		delete [] config.bitstream;
	}
}

DWORD *readBitFile (const char *filename, int *numBytes)
{
	FILE *		inFile;
	unsigned char *	buffer;
	long				fileSize;

	// try to open the input file
	if ((inFile = fopen (filename, "rb")) == NULL) {
		*numBytes = 0;
		printf ("readBitFile: Couldn't open file '%s\n", filename);
		return NULL;
	}

	// figure out how long the file is
	fseek (inFile, 0, SEEK_END);
	fileSize = ftell (inFile);
	*numBytes = (int) fileSize;
	// make sure numBytes is accurate...
	if (*numBytes != fileSize) {
		printf ("readBitFile: Oh, no!  File '%s' is bigger than an
	int!\n", filename);
		fclose (inFile);
		return NULL;
	}

	// make the new buffer
```

```
        buffer = new unsigned char [fileSize];

        // read in the bytes
        fseek (inFile, 0, SEEK_SET);
        fread ((void *) buffer, sizeof (unsigned char), fileSize, inFile);
        return (DWORD *) buffer;
}
```

Improving the Performance and Efficiency of an Adaptive Amplification Operation Using Configurable Hardware *

Michael J. Wirthlin[1], Steve Morrison[1], Paul Graham[1], and Brian Bray[2]
[1]Department of Electrical and Computer Engineering, Brigham Young University, Provo, Utah 84602
[2]Sandia National Laboratories, P.O. Box 5800, MS 0844, Albuquerque, NM 87185

1 Abstract

An adaptive amplification operation has been designed and tested in configurable hardware for a computationally intensive object recognition system. This configurable system provides over forty-one times the throughput of an industry-standard embedded processor by exploiting the bandwidth of internal block memories and parallelism within the algorithm. Operating at less than one half the power of the programmable processor, the configurable approach performs the computation with 90 times less energy. The improvements in both performance and power are obtained by customizing the datapath, memory interfaces, and control to the amplification algorithm.

2 Introduction

Configurable computing approaches are increasingly used in embedded systems to accelerate application-specific computing tasks. This technology is frequently used in signal processing systems or sensor interfaces that require real-time processing of high-bandwidth interfaces [1, 2]. For some environments, configurable computing approaches have replaced high-performance embedded processors to provide greater throughput at a lower system cost. In addition, these configurable systems preserve the programmability of a system by allowing upgrades, improvements, or multiple application-specific circuits through device reconfiguration.

Configurable systems may also perform application-specific functions with greater efficiency than general-purpose programmable architectures [3, 4]. The ability to customize the datapath, memory interfaces, and control eliminates much of the overhead found in programmable processor architectures. Improvements in computational efficiency suggest configurable computing approaches may perform certain computations with significantly less power than a conventional programmable processor [5, 6].

This paper will present an adaptive amplification operation and demonstrate how configurable hardware performs this operation with 41 times more throughput than the current microprocessor approach. Further, the configurable approach performs this operation with less than one half the power dissipation of the programmable processor for a 90 times improvement in energy consumption per operation. The improvements in both performance and power suggest that configurable hardware approach is a superior alternative for high-performance power-sensitive, embedded environments.

The paper will commence with a brief description of the adaptive amplification algorithm and discuss its use within a pattern recognition system. Next, the current implementation using a low-power embedded processor will be introduced and analyzed. The configurable hardware alternative will then be introduced and contrasted with the more conventional programmable processor approach. Finally, the improvements in efficiency gained with custom hardware will be demonstrated by comparing the power dissipation of both approaches.

3 Adaptive Amplification

Automatic target recognition (ATR) systems often involve several computationally intensive image-processing algorithms. Such systems are required to perform these algorithms at extremely high data rates (i.e. real-time image data). The throughput requirements and computational complexity of such

*This effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

algorithms frequently exceed the capabilities of conventional programmable processors and require the use of custom hardware. Configurable computing approaches have been identified and implemented to solve several such ATR algorithms [7, 8].

One such ATR system is currently being developed by Sandia National Laboratories for the U.S. Department of Defense Joint STARS airborne radar imaging platform. Like most ATR systems, the ATR for the Joint STARS platform performs multiple passes on potential target data to refine the confidence of a pattern match and filter unnecessary data. Coarse recognition algorithms are used to filter potential targets for more computationally intensive recognition algorithms downstream.

A key component of this process is an adaptive amplification operation. This operation must dynamically determine the gain of a potential target signal before further processing. By dynamically adapting the gain of the input signal, this pattern matching system is much more successful at identifying targets in spite of signal noise or obstruction. The input signal of Figure 1 demonstrates the need for dynamic gain control. This signal is attenuated by some unknown source and includes a signal obstruction. Due to the attenuation, the signal falls outside of the target reference upper and lower signal bounds. Unless this signal is amplified to fall within these bounds, the target will pass undetected. This adaptive amplification operation will dynamically identify the optimal gain needed to insure that such input signals fall within the reference bounds.

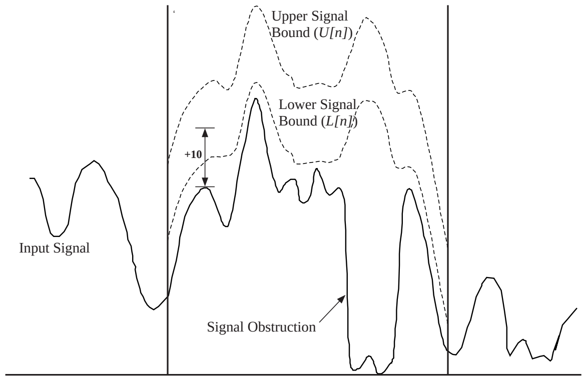


Figure 1: Sample Input and Reference Signals.

Unlike many signal processing algorithms, this algorithm does not operate on continuous input data. Due to its computational requirements, this algorithm operates on finite sequences of potential target data identified by a previous pattern recognition operation. The adaptive amplification algorithm must determine the optimal gain at each location within this search region. The gain values will be used by subsequent algorithms in the pattern recognition system.

The algorithm determines the gain value by examining a finite window within the search region (see Figure 2). A dynamic gain value, $g[n]$, is computed at *each* location within the search region using a finite window of N samples (i.e. using samples $x[n-N+1]$ through $x[n]$). The optimal gain value is chosen to maximize the number of samples that fall within the target bounds, or maximize the following value:

$$\sum_{k=0}^{N-1} b(x[n-N+k]+g[n],k), \text{ and} \tag{1}$$

$$b[i,j] = \begin{cases} 1 & \text{if } L[j] \leq i \leq U[j] \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where $U[j]$ and $L[j]$ represent the upper and lower target signal bounds. Using this criteria for optimal gain control, an offset of +10 of Figure 1 will maximize the number of input samples that fall within the bounds of the target pattern.

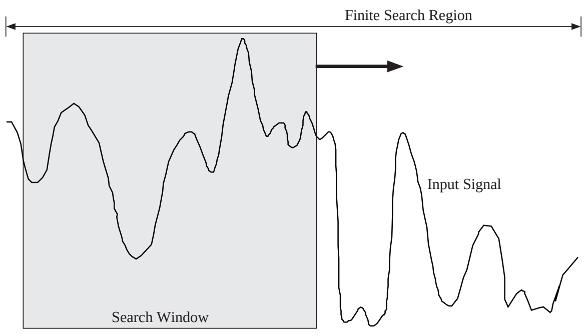


Figure 2: Search Region.

Determining the optimal gain at each sample involves two operations. First, each sample within the window is compared against the corresponding upper and lower signal bounds to create a statistical distribution of potential gain values. Second, the distribution is searched to identify the gain value that maximizes the number of input samples that fall within the signal bounds. This optimal gain value is then used to scale the samples of the input window before other target recognition operations. Both phases of this operation will be described in more detail below.

3.1 Creating Distribution of Gain Values

As suggested in Equation 2, each input sample in the finite window ($x[n - N + k]$) must be compared against the corresponding Upper ($U[k]$) and Lower ($L[k]$) target signal bounds. Subtracting an input sample from the corresponding lower reference sample provides a lower bound ($R_l[n, k]$) on the gain value needed to place the sample within the pattern range. Subtracting the input sample from the corresponding upper reference sample provides an upper upper bound ($R_u[n, k]$) on the gain value allowable to place the sample within the pattern range. Any offset value within the range defined by these bounds (i.e. $R_l[n, k] \leq g[n] \leq R_u[n, k]$) will place the input sample $x[n - N + k] + g[n]$ within the upper and lower bounds of the reference signal. For an input window of N samples, N unique ranges are computed.

A histogram of gain values is created by accumulating the N ranges of gain values. Each bin in the histogram represents a unique gain value – the bin with the highest bin count represents the gain value that maximizes the number input samples that lie within the upper and lower references signals. Figure 3 demonstrates a histogram table created from 5 offset ranges ($N = 5$). The optimal offset value is +6.

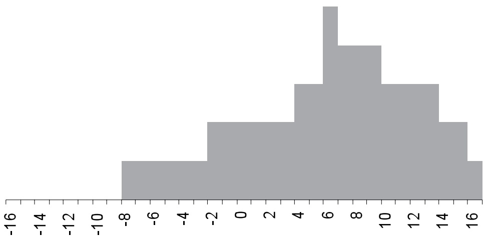


Figure 3: Sample Signal Offset Histogram

Creating a histogram this way can be a costly operation. For a range of values defined by $u[n - R_l] - u[n - R_u]$ (where $u[n]$ represents the unit function), $R_u - R_l + 1$ histogram accesses are required. Histogram table accesses can be reduced by accumulating the derivative or *difference* of each finite range:

$$(u[n - R_l] - u[n - R_u])' = \delta[n - R_l] - \delta[n - R_u] \quad (3)$$

The derivative of a finite range contains only *two* samples (+1 at R_s and -1 at R_e) no matter how large the range is. Accumulating the derivative of each range produces a table that contains the derivative of the desired histogram. Figure 4 demonstrates the derivative of the histogram shown in Figure 3. The original histogram can be reproduced by by integrating this derivative table.

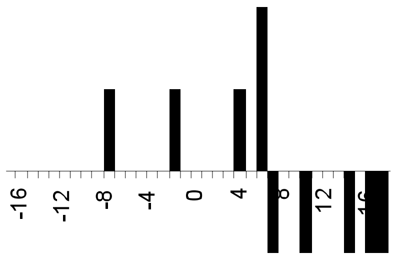


Figure 4: Sample Histogram Derivative

Building this histogram table requires only a few simple arithmetic operations. Two subtractions are needed to obtain the indices of the table update, six comparison operations are needed to insure the indices fall within the allowable gain range and an incrementer/decrementer pair are needed to update the corresponding values of the table. The pseudo-code for this simple operation is shown in Listing 1.

```
for (i=1 to N) {
  lower_index = L[i] - x[n-N+i];
  if (lower_index >= TABLE_START &&
      lower_index <= TABLE_END)
     histogram[lower_index]++;
  upper_index = U[i] - x[n-i];
  if (upper_index >= TABLE_START &&
      upper_index <= TABLE_END)
     histogram[end_index]--;
  if (lower_index < TABLE_START &&
      upper_index >= TABLE_START)
     below_table_count++;
}
```

Listing 1: Histogram Create Operation.

Clearly, creating this table does not require complicated arithmetic operators. However, several memory accesses are required to obtain the data needed for this computation. One memory access is needed to retrieve the input signal, two accesses are needed to obtain the upper/lower target values, and up to four accesses are necessary for updating two histogram locations (a histogram update requires a read followed by a write). In total, seven memory accesses are needed to update the table for each input sample.

3.2 Searching the Histogram Table

Once the table has been created, the next stage of the operation is to search the table for the optimal gain value. Since the table contains the derivative of the histogram, the table must also be integrated to reconstruct the original histogram. Like the table create phase, this step in the algorithm can be performed with simple arithmetic operations. Specifically, only two arithmetic operations are needed for each bin of the histogram — an addition to perform the integration and a comparison to identify the optimal gain value. This table integration and searching function can be combined as shown in Listing 2.

```
current_sum = below_table_count;
max_sum = 0;
for (i=0; i<histogram_size; i++) {
  current_sum += histogram[i];
  if (current_sum > max_sum) {
    max_sum = current_sum;
    max_index = i;
  }
  histogram[i]=0;
}
```

Listing 2: Histogram Integration and Search.

Assuming `max_sum`, `max_index` and `current_sum` are in local registers, only two memory access are required for each bin in the table. One access is required to read the histogram bin and the other is required to clear the bin for the next pass of this operation.

3.3 Complexity

The complexity of the operation depends upon the number of input samples used to create the histogram (N) and the number of the histogram bins (S). The number of arithmetic operations and memory accesses required for this algorithm is described in Table 1 in terms of N and S. This study will assume an input window with N=128 samples and a histogram with S=64 bins.

4 Programmable Processor Implementation

The adaptive amplification algorithm described above currently operates in an embedded environment where power, form-factor, and system costs limit the implementation options. The current implementation approach for this project uses the 300 MHz Motorola PowerPC 603e[9], a low-power embedded variant of the popular PowerPC RISC architecture [10]. Although other more powerful and faster programmable processors are available, this processor is used because of its low-power and on-chip system resources.

	Memory Accesses	Arithmetic Operations
Table Create	$7N$	$9N$
Table Search	$2S$	$2S$
Total	$7N + 2S$	$9N + 2S$

Table 1: Complexity of Adaptive Amplification Operation.

This RISC architecture employs most modern microprocessor features such as multiple instruction issue, out-of-order execution, and branch prediction. Five execution units are available to this superscalar processor — an integer unit (IU), a floating-point unit (FPU), a load/store unit (LSU), a branch processing unit (BPU), and a system register unit (SRU). In addition, this microprocessor contains several system components that reduce the need for many system interface devices.

The adaptive amplification operation described in Listings 1 and 2 was written in 'C' and compiled to this microprocessor using the gnu C compiler (gcc). The algorithm was simulated on the PSIM cycle-accurate instruction set simulator developed for the PowerPC family of microprocessors [11]. This simulator provides instruction tracing, pipeline status, and detailed timing information from user simulations.

The simulator was used to determine the performance of this adaptive amplification operation executing on the microprocessor. Representative reference and input data were used in the simulation with N = 128 (window size) and S = 64 (histogram size). The results from several simulations are summarized in Table 2. These results provide the performance of the microprocessor for both phases of the adaptive amplification operation. As shown in the table, the average execution time of this operation on microprocessor is 12.9 *us* for a throughput of 77,500 samples/second.

It is interesting to note that the operation does not take advantage of the multiple execution units within the microprocessor. The sustained instruction throughput of this program is only 1.10 instructions per cycle. The instruction-level parallelism is limited by the data dependencies and frequent memory ac-

	Table Create	Table Search	Total
Instructions	3517	742	4259
Clock Cycles	3323	543	3866
Instructions per cycle (IPC)	1.06	1.37	1.10
Execution Time	11.1 *us*	1.8 *us*	12.9 *us*

Table 2: Execution Summary of Adaptive Amplification Operation Executing on 300 MHz PPC-603e.

cesses found in the algorithm. Since few operations are performed for each data access, the integer and load/store unit are not able to operate in parallel. The data dependencies between instructions and pipelining stalls due to branching and memory accesses eliminate the positive effects of instruction-level parallelism.

5 FPGA Implementation

Although the PowerPC provides relatively high-performance in such a small form factor, it does not offer enough performance to meet the real-time computational requirements of the pattern recognition system. Reconfigurable computing technology is an important alternative that may be used in the computation of this and other real-time signal processing systems. The ability to exploit custom functional units, control, and memory access patterns offers an efficient alternative to traditional programmable processors.

For this operation, where numerous memory accesses are made to the global histogram table, internal FPGA memories are used to provide efficient on-chip memory. This adaptive amplification operation has been implemented using a relatively simple design based on the Virtex FPGA manufactured by Xilinx [12]. The circuit is centered around a Virtex Block RAM which is used to hold the contents of the histogram table. Surrounding circuitry is added to maximize the efficiency of the histogram table access.

As described earlier, there are two phases of operation — table create and table search. Custom circuitry is used for each phase to address and manipulate the data within the memory. The circuitry for each phase will be described below.

5.1 Table Create

During histogram build mode (see Listing 1), both independently addressable ports of the histogram block RAM are used. One port is dedicated to increment the "start of range" bin and the other to decrement the "end of range" bin. A dedicated incrementer and decrementer is attached to the outputs of these ports. This allows the increment and decrement operations to occur simultaneously.

Two dedicated subtractors are used to address the two ports of the histogram table. These subtractors simultaneously compute the indices for two bins in the table — the start of the range $(L[i] - x[n-i])$ and the end of the range $(U[i] - x[n-i])$. These two subtractors also perform the overflow and underflow detection. If the results of either subtract produce an index that falls outside of the histogram range, the overflow/underflow circuitry will disable the memory port to prevent the corresponding increment or decrement. A block diagram of the circuitry used to create the table is shown in Figure 5.

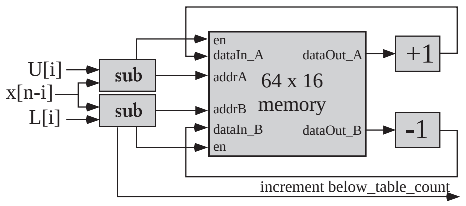


Figure 5: Block RAM Histogram Table

Two clock cycles are required to update a pair of histogram bins — one cycle to read both bin values and another cycle to update the bins with new results. By supplementing the block memory with dedicating addressing and computational logic, both memory ports are active during *each* cycle of the computation. This custom circuit will build the entire histogram using only $2N$ clock cycles.

5.2 Table Search

During table search mode (see Listing 2), a simple address counter is used to sequence through each of the S bins in the histogram table. Again, both memory ports are used simultaneously — one port is used to sequentially access each bin in the table for integration/searching and the other port is used for clearing the table in preparation for the next window of samples.

A custom circuit attached to the histogram table memory integrates the table bin values and identifies the bin with the highest probability. This integration and searching circuit includes an adder, comparator, and registers and is shown in Figure 6. Only one cycle is required to access, integrate, and clear each bin in the table. The complete integration and search operation consume S clock cycles.

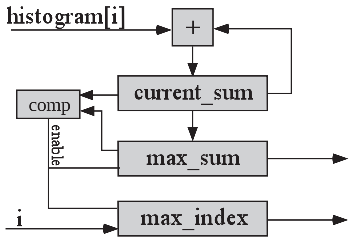


Figure 6: Search Circuit

The designs depicted in Figures 5 and 6 were mapped to the Xilinx Virtex device. The circuit consumes one block RAM and 104 Virtex slices. The technology mapping tools estimate the maximum frequency at 80 MHz[1].

6 Performance Comparison

The performance results of both the PowerPC and FPGA implementation are contrasted in Table 3. As seen in the table, the FPGA implementation provides over three times the throughput of the embedded processor. More striking is the fact that the FPGA achieves this performance at *one fourth* the clock rate of the embedded processor. This suggests that the custom hardware solution performs 12 times more computation during each clock cycle than the embedded processor.

Clearly, the custom hardware performs this operation much more efficiently than the microprocessor. The custom hardware approach achieves significantly more computation per clock cycle by dedicating custom hardware for each arithmetic operation, providing dedicated memory ports for each memory access, and synchronizing the operations using a statically scheduled custom controller.

There are several reasons the microprocessor is less efficient than the custom hardware. First, the microprocessor is operating in a sequential model of computation. Even though multiple execution units are available for this operation, the data dependencies imposed by the sequential programming model restricts the amount of parallelism (this operation averages 1.1 instructions/cycle). Second, large and complex support circuitry are needed to insure a rapid and consistent instruction execution rate. The large on-chip caches, branch prediction units, pipelined execution units and complex instruction sequencing units require far more hardware than the 18,500 estimated “gates” of the custom hardware solution.

[1]The circuit was mapped to the XCV1000 -6 device in the BG560 package

	PowerPC	FPGA	Speedup
Clock Rate	300 MHz	80 MHz	.267
Table Create (clock cycles)	3323	256	13x
Table Search (clock cycles)	543	64	8.5x
Total (clocks)	3866	320	12x
Total (time)	12.9 *us*	4 *us*	3.2x
Throughput (samples/sec)	77,500	250,000	3.2x

Table 3: Performance Comparison of Embedded Processor and FPGA.

7 Parallel Approach

Although the custom circuit used to implement this algorithm provides greater performance than a programmable processor, it probably does not provide sufficient advantages to justify the use of a CCM solution. A CCM used in this system would utilizes only a small fraction of the FPGA resources on the board. However, there is parallelism in the algorithm, and it is possible to exploit this parallelism by using additional FPGA resources. If this parallelism can be exploited by the CCM hardware, it is likely that a CCM approach will provide enough performance improvement to justify its use in this target recognition system.

As suggested in Figure 7, this algorithm operates on multiple locations within a search region. Since the same table create and table search operations must be performed at each search location, it is possible to perform this adaptive amplification algorithm for multiple search locations in parallel. With sufficient hardware, the algorithm could operate simultaneously on all six search locations shown in Figure 7.

To operate on more than one search location in parallel, a unique histogram table (i.e. block RAM) is needed for each concurrent search location. In addition, each histogram table requires the build and search circuitry shown in Figures 5 and 6. Ideally, a

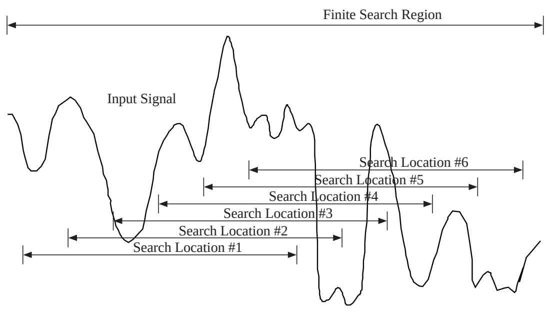


Figure 7: Multiple Search Locations.

linear speedup is achieved for each processor replicated in the hardware.

As with any parallel approach, additional hardware resources and memory bandwidth are needed to implement each additional adaptive amplification circuit. Fortunately, the logic resources of this circuit are nominal – many such circuits can be placed on todays larger FPGA devices. However, the custom circuits used for this algorithm require a considerable amount of bandwidth. Unless the bandwidth to a set of adaptive amplification circuits is carefully managed, only limited parallelism can be exploited by the circuit.

7.1 Bandwidth Requirements

There are two bandwidth issues that must be addressed by a parallel implementation of this circuit – the internal bandwidth required by each adaptive amplification processor and the external bandwidth between the CCM and the host. As suggested in Section 5, two reads and two writes are required every two clock cycles to update the histogram table. At 80 MHz, 160 MB/sec of internal bandwidth is required for *each* processor. However, it is important to note that this internal bandwidth can be replicated by using a custom block RAM for each processor. So long as there are enough internal block RAMs for each processor, the internal bandwidth issue does not present a problem.

The second bandwidth issue that must be addressed is the external bandwidth between the CCM and the host. Each processor requires 3 bytes every two cycles – one byte each for the upper bound value, the lower bound value, and the image pixel. Thus, to feed each processor without stalling, 120 MB/sec of data is necessary or $120L$ MB/sec for L processors. The bandwidth requirements quickly limit the parallelism that can be extracted from the algorithm. For example, if a 64-bit data path is provided by the CCM (640 MB/sec at 80 MHz) only 5 processors would be usable.

Two approaches are used to reduce the external bandwidth problem. First, the upper and lower data can be preloaded into internal block RAMs. Since each processor is acting on data from the same template, the same upper and lower bound functions are used for every location within the search region. Thus, the upper and lower values can be used simultaneously by all of the parallel processors. By pre-loading it into block RAMs, the data can be re-used without a strain on the external memory bus.

The second solution is to exploit the locality of input data. As shown in Figure 2, adjacent search regions operate on overlapping data. Rather than refetch the image information from memory each time it is needed, a bank of registers was created to store the data and reuse it for a different search region.

7.2 Parallel Implementation

For the Sandia ATR system, the algorithm must search 15 regions for each potential target sample. 15 adaptive amplification processors can operate in parallel to compute the optimal gain for each of the 15 search locations simultaneously. An architectural view of this approach is shown in Figure 8.

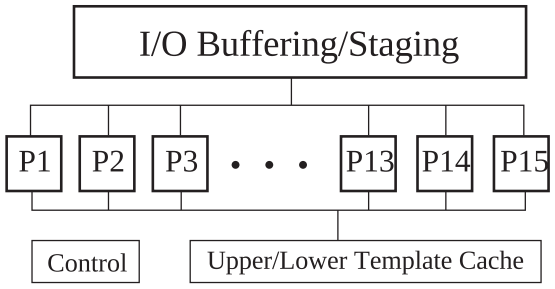


Figure 8: Parallel Adaptive Amplification Processors.

The 15 processors, I/O interfaces, and control consume 2090 FPGA slices and 18 Block RAMS[2]. Although this circuit consumes considerably more resources than the single adaptive amplification processor described in Section 5, it easily fits on a Xilinx XCV400 device. To verify this implementation approach, the circuit was designed and tested on the Annapolis Microsystems Wildstar board (using the XCV1000 FPGA).

176 125 0 3 51,715

[2]The Xilinx technology mapping tools suggest these resources equate to 330,000 gates.

7.3 Performance of Parallel Implementation

Placing fifteen processors in parallel approaches a 15X speedup of the single-processor circuit. However, the additional overhead necessary for filling the template cache and initiating the I/O interfaces will limit a linear performance improvement. The overhead associated with these cycles increases the time for the processors to complete a search by 16%. With 15 processors operating in parallel, the average search time for a region is 24.7 clock cycles for a 41 times improvement in performance over the programmable processor. Table 4 summarizes the performance of this parallel circuit.

	PowerPC	FPGA	
Processors	1	1	15
Clock Cycles	3866	320	372
Clock Rate	300 MHz	80 MHz	80 MHz
Time (*us*)	12.9	4.0	4.65
Area (slices)	N/A	104	2090
Throughput (Ksamples/sec)	77.5	250	3125
Speedup	1.0x	3.2x	41.2x

Table 4: Performance Comparison of Processor and Parallel CCM Approach.

As shown in Table 4, the parallel CCM solution provides over 40 times more throughput than the programmable processor. The CCM approach is able to achieve this performance by carefully managing the internal bandwidth of the FPGA. The 15 processors, along with the template cache, consume an aggregate internal bandwidth of 2,480 MB/sec. This bandwidth is comparable to the bandwidth available at the L1 cache of the PowerPC. However, the custom hardware approach utilizes its memory bandwidth more efficiently by carefully scheduling each access and performing the arithmetic operations in parallel.

8 Power

Another way to evaluate the computational efficiency of each implementation approach is to compare the average power consumed during the execution of this adaptive amplification operation. Ideally, the power consumption of both approaches would be measured directly from each system during execution. However, since the actual systems are not readily available, power *estimates* will be provided. These estimates are based on the appropriate datasheets and power estimate worksheets.

The datasheet for the PowerPC 603e states that the *average* power dissipation for this processor is 4.4 W [13]. The power dissipation of the FPGA solution was estimated using the Virtex Power Estimator available from Xilinx [14]. This worksheet requires design parameters such as the clock rate, design size (slices, BlockRAMs, etc.), and average toggle rates. The design size and clock rate are taken directly from the technology mapping report file. The average toggle rate for each the design was estimated through design simulations using the JHDL design tool [15]. Based on the information provided to this worksheet for this adaptive amplification operation, the average power dissipation is estimated at 2.0 W.

Although these power estimates are approximate, they suggest that the configurable hardware performs this adaptive amplification computation using less than half the power required by the microprocessor (see Table 5). This result appears reasonable — the configurable approach operates at one fourth the clock rate of the microprocessor and uses a modest amount of logic resources.

The efficiency of the configurable approach is also shown by calculating the energy required to perform the computation for a single sample. Since the parallel adaptive amplification circuits perform the computation 41 times faster than the microprocessor and requires 2.2 times less power, it requires *90 times* less energy than the microprocessor to perform the computation for a single set of samples (see Table 5).

	PowerPC	FPGA	
Average Power	4.4W	2.0W	2.2x
Time (per sample)	12.9 *us*	312 *ns*	41.2x
Energy (per sample)	57 *uJ*	.620 *uJ*	92x

Table 5: Estimated Power and Energy Consumption.

9 Conclusion

A custom circuit was designed with configurable hardware to improve the performance of an adaptive amplification operation. This approach was analyzed and compared against the existing solution —

an industry standard embedded processor. This configurable solution provides greater performance than the microprocessor by using customized datapath, optimized memory interfaces, and statically scheduled control. The performance of the microprocessor is limited due to data dependencies between instructions, flow-control overhead, and limited opportunities for instruction-level parallelism.

The improvements in computational efficiency gained by custom hardware also reduces the power requirements of the operation — the configurable solution performs the operation faster and using almost two orders of magnitude less energy than the microprocessor. Based on the positive results of this preliminary investigation, the configurable solution has been identified as an attractive alternative to the embedded microprocessor.

References

[1] M. Shand. Flexible image acquisition using reconfigurable hardware. In P. M. Athanas and K. L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 125–134, Napa, CA, April 1995.

[2] T. Moeller and D. R. Martinez. Field programmable gate array based radar front-end digital signal processing. In K. L. Pocek and J. M. Arnold, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 178–187, Napa, CA, April 1999.

[3] André DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, September 1996.

[4] P. Graham and B. Nelson. FPGA-based sonar processing. In *Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '98)*, pages 201–208. ACM/SIGDA, ACM, 1998.

[5] Arthur Abnous, Katsunori Seno, Yuji Ichikawa, Marlene Wan, and Jan Rabaey. Evaluation of a low-power reconfigurable DSP architecture. In *Proceedings of the Reconfigurable Architectures Workshop*, March 1998.

[6] Jan M. Rabaey. Reconfigurable computing: The solution to low power programmable DSP. In *Proceedings of 1997 ICCASP Conference*, 1997.

[7] J. Villasenor, B. Schoner, K. Chia, and C. Zapata. Configurable computing solutions for automatic target recognition. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 70–79, Napa, CA, April 1996.

[8] M. Rencher and B. Hutchings. Automated target recognition on Splash 2. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 192–200, Napa, CA, April 1997.

[9] Motorola. *PowerPC 603e RISC Microprocessor Family: PID7t-603e Hardware Specifications*, 1999. Advance Product Specification.

[10] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors.* Morgan Kaufmann Publishers, 1994.

[11] Andrew Cagney. Psim – model of the powerpc architecture. http://sourceware.cygnus.com/psim/, April 1996.

[12] Xilinx Corporation. *Virtex 2.5 Field Programmable Gate Arrays*, May 1999. Advance Product Specification.

[13] Motorola. *PowerPC 603e RISC Microprocessor Family: PID7t-603 Hardware Specifications*, 1/1999. MPC603E7TEC/D, REV. 3.0.

[14] Xilinx Corporation. *Virtex Power Estimator User Guide*, April 28 1999. XAPP 152 (Version 1.0).

[15] Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, and Mike Rytting. A CAD suite for high-performance FPGA design. In K. L. Pocek and J. Arnold, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1999.

Matching Genetic Sequences in Distributed Adaptive Computing Systems

William J. Worek

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

July 29, 2002
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

Keywords: configurable computing, FPGA, adaptive computing, genetic sequences, pattern matching

Matching Genetic Sequences in Distributed Adaptive Computing Systems

William J. Worek

Abstract

Distributed adaptive computing systems (ACS) allow developers to design applications using multiple programmable devices. The ACS API, an API created for distributed adaptive computing, gives developers the ability to design scalable ACS systems in a cluster networking environment for large applications. One such application, found in the field of bioinformatics, is the DNA sequence alignment problem. This thesis presents a runtime reconfigurable FPGA implementation of the Smith-Waterman similarity comparison algorithm. Additionally, this thesis presents tools designed for the ACS API that assist developers creating applications in a heterogeneous distributed adaptive computing environment.

To Everyone,

so I don't leave out anyone.

Acknowledgements

While working on my thesis, a number of people have provided an enormous amount of support. First, I would like to thank my committee, Dr. Athanas, Dr. Baker, and, especially, my advisor Dr. Jones. A debt of gratitude is owed to all of the people that had a hand in developing the ACS API. A special thanks goes to Kiran Puttegowda and James Hendry. Without their support, none of this would have been possible. I would also like to thank all of the diligent workers in the CCM Lab for providing me with entertainment as well as guidance. I would like to acknowledge DARPA and ISI for supporting the development of this project. A many thanks go to all of the people that have helped me get to this stage in my life. And finally, I would like to acknowledge the support of all of my friends and family for all of their help.

William J. Worek

July 2002

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

Since the introduction of the Human Genome Initiative [1], genetic sequences are being discovered at an ever increasing rate. It is a common problem in the field of bioinformatics to compare a gene sequence to a database of sequences, or databases of sequences against another database. Gene repositories, such as GenBank [2] or Swiss-Prot [3], used in these comparisons have grown to be quite large and require a great deal of computation to compare against. Faster methods of comparing gene sequences are required in order to keep up with the size of the databases.

Developing large distributed applications on adaptive computing systems requires knowledge of hardware design languages, FPGA board APIs, and a method for communicating between computers. Becoming proficient in all of these areas is a difficult task. Furthermore, there are few resources for debugging and developing applications in a distributed adaptive computing environment.

1.2 Thesis Statement

This thesis explores the use of the Adaptive Computing System API (ACS API) for developing a distributed gene matching application. This thesis contributes the following.

- It demonstrates and analyzes the use of the ACS API in developing distributed systems.
- It suggests methods for designing distributed adaptive computing systems.
- It analyzes the results of using a runtime reconfigurable approach to comparing genetic sequences on FPGAs.
- It demonstrates and analyzes the use of heterogenous nodes in an adaptive computing system.

A runtime reconfigurable gene matching implementation of the Smith-Waterman [4] algorithm was developed as part of this research. To assist in developing distributed adaptive computing systems, a textual and graphical interface was created for the ACS API.

1.3 Thesis Organization

This thesis is organized in the following manner. Chapter 2 presents the background for the work presented in this thesis. The background discusses related research and concepts that contributed to this thesis. Chapter 3 describes the ACS API. It explains the development environment that was used in designing the distributed systems. Chapter 4 describes the user interface designed to assist in developing applications. Chapter 5 presents the hardware implementation of the genetic sequence alignment algorithm. Chapter 6 describes the distributed adaptive computing systems developed to utilize the hardware described in Chapter 5. Chapter 7 presents an analysis of the results from the systems developed. Chapter 8 summarizes the research and proposes avenues for future work.

Chapter 2

Background

This section will discuss the background information used in this thesis. The background is divided into four sections. The first section discusses adaptive computing devices and examines the resources available on two boards used. The second section explains the Smith-Waterman [4] pattern matching algorithm. The third section gives an overview of the genetic code matching problem, as well as examining alternative algorithms and commercial implementations. The final section examines applications that make use of pattern matching.

2.1 Adaptive Computing Devices

Adaptive Computing Systems (ACS) are comprised of reprogrammable hardware devices. These hardware devices contain programmable logic devices (PLDs) such as commercially available Xilinx Field Programmable Gate Arrays (FPGAs). This section gives an overview of PLDs. This section also gives a description of the SLAAC-1V and the Osiris, the two Xilinx-based platforms used in this thesis.

2.1.1 Overview

Digital logic devices are typically divided into two broad categories, Application Specific Integrated Circuits (ASICs) and general purpose processors (GPPs). GPPs, such as CPUs in a personal computer, are very flexible devices that are characterized by a high degree of programmability at the sacrifice of performance. The GPPs limited performance is due to the overhead of decoding and executing instructions. ASICs are integrated circuits designed for a specific task. In contrast to GPPs, a very high level of performance and limited flexibility defines an ASIC. ASICs may have a limited amount of flexibility that is built into the circuit during the design. Any flexibility typically results in a sacrifice in performance [5].

Programmable Logic Devices (PLDs) allow a high degree of flexibility without sacrificing very much in performance. PLDs are approximately three times slower than ASICs, but can be significantly faster than GPPs [5]. A typical PLD, such as a Xilinx FPGA, is comprised of three parts, Input/Output Blocks (IOBs), Configurable Logic Blocks (CLBs), and a programmable interconnection network. IOBs are interfaces for connections external to the chip. A large array of CLBs implements the desired logic. The programmable interconnection controls signal routing between CLBs [6].

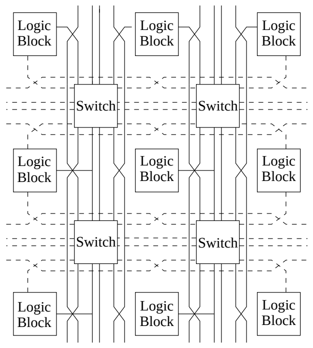


Figure 2.1: Basic FPGA Components

The CLB consists of function generation, internal routing, and memory. To generate functions, the CLB uses an array of lookup tables (LUTs). The internal routing allows signals to be connected to the memory or continue straight out of the CLB. D-flip-flops (DFF) provide fast memory for storing intermediate values. The ability to store values local to the CLB allow a high degree of pipelining in the circuit. The CLBs and interconnections are programmed using anti-fuses or memory. Anti-fuses are one-time programmable cells that create permanent connections when stimulated with a current. Modern PLDs use memory units such as Static RAMs (SRAMs) to program the device. Figure 2.1 shows logic blocks (LB) in relation to programmable switch matrices (PSM). A configuration defines the hardware which the PLD implements by specifying the values in all of the LUTs and SRAMs used in the CLBs and programmable interconnections [5].

2.1.2 SLAAC-1V

The Information Sciences Institute (ISI) East, a part of the University of Southern California (USC), developed the SLAAC-1V under the Systems Level Application of Adaptive Computing (SLAAC) project [7]. The Defense Advanced Research Project Agency (DARPA) sponsored the SLAAC project, which combined the resources of several different organizations including the Configurable Computing Machines Laboratory of Virginia Tech.

The SLAAC-1V [8] is a FPGA platform capable of being connected to a host computer via a 32-bit 33 MHz PCI connection. Three Virtex 1000 FPGAs reside on the SLAAC-1V board. The Virtex XCV1000 FPGAs contain the equivalent of 1 million system gates and 27,648 Logic Cells. Each part hosts 64x96 CLBs and 128K bytes of Block RAM. Each CLB in the Virtex chip is composed of two slices. A slice is the most primitive logic block contained in an FPGA, made up of 2 DFF, 2 4-input LUTs, and assorted logic for internal routing [6]. The three FPGAs are named X0, X1, and X2. The X0 FPGA contains the interface to the CPU, known as the Interface FPGA (IF). For convenience, X0 and IF are referred to as

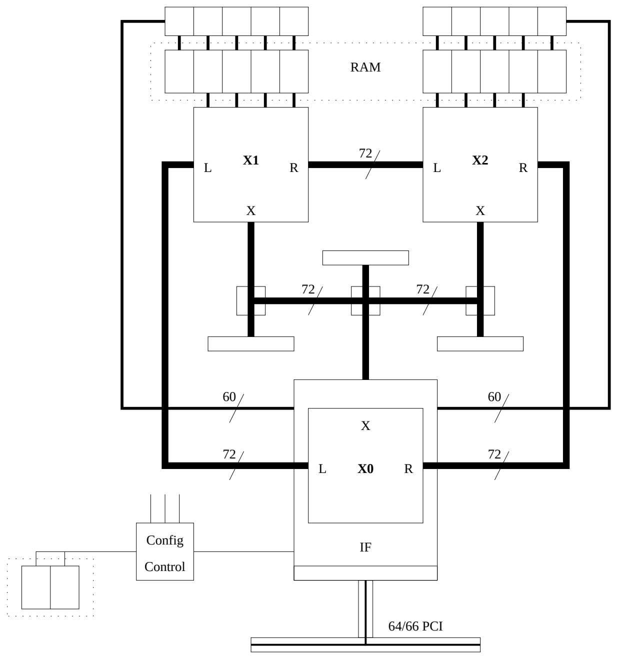


Figure 2.2: SLAAC-1V Board Architecture

separate units within the SLAAC-1V.

The SLAAC-1V also contains memory on the board. The user FPGAs X1 and X2 can each access four zero bus turnaround (ZBT) SRAMs. The X0 FPGA only has two ZBT SRAMs available to it. All the ZBT SRAMs have 36 bit words and are 256K addressable. ZBT RAMs allow a read or write during every clock cycle, allowing 100 percent utilization of the bus. The user can specify whether the IF will preempt the FPGAs to allow the host access to the memories. During preemption, the clocks on the FPGAs are halted while the memories are being used. The user can also specify that the IF use non-preemptive memory access. In this case, the user must ensure that the host and the FPGAs do not access the memories at the same time [8].

The SLAAC-1V has two sets of FIFOs. FIFO A0 is an input to the board and has a complement output FIFO named B0. Likewise, FIFO A1 is used for input and FIFO B1 is used as an output. The A0/B0 FIFO is 64 bits wide, but has no buffering. This type of FIFO is referred to as a mailbox. The A1/B1 FIFO also has words which are 64 bits wide. In contrast to the mailbox FIFOs, the second type of FIFOs can buffer up to 256 words [8].

A crossbar exists between X0, X1, and X2. The crossbar is configured by the X0 element and allows data to be broadcast between the processing elements. In addition to the crossbar, there is a 72-bit connection that creates a ring. The ring connection, using RIGHT and LEFT descriptors, is useful for streaming data through all of the processing elements. Figure 2.2 shows the architecture of the SLAAC-1V board.

2.1.3 Osiris

The Osiris board [9] is the latest generation from the SLAAC family of FPGA boards. Like its predecessors, the Osiris board was developed at ISI-East under the SLAAC project sponsored by DARPA. The Osiris board has many improvements in hardware over the SLAAC-1V. The design of the drivers and support software has also changed.

The Osiris board contains two Virtex II FPGAs. One FPGA, the IF, is used as an interface between the FPGA board and the host computer. The interface FPGA is a Virtex II 1000 (XC2V1000) part. The second FPGA, the XP, is a user programmable device available for implementing hardware designs. The user FPGA is a Virtex II 6000 (XC2V6000) part [9].

The Virtex II 1000 device contains the equivalent of one million system gates. A total of 720K of block RAM bits reside on this device. In contrast, the Virtex II 6000 is a much larger device, containing the equivalent of six million system gates. A total of 2,592K block RAM bits are contained in this device. The number of configurable/complex logic blocks (CLBs) increase from 40x32 in the 1000 part to 96x88 in the 6000 part [10].

The Osiris board contains two types of memory available to the XP and two types of memory accessible to the IF. The memory attached to the IF is typically used for holding configurations or partial reconfigurations. Four megabytes of flash memory is available to the IF along with six megabytes of asynchronous SRAM [9].

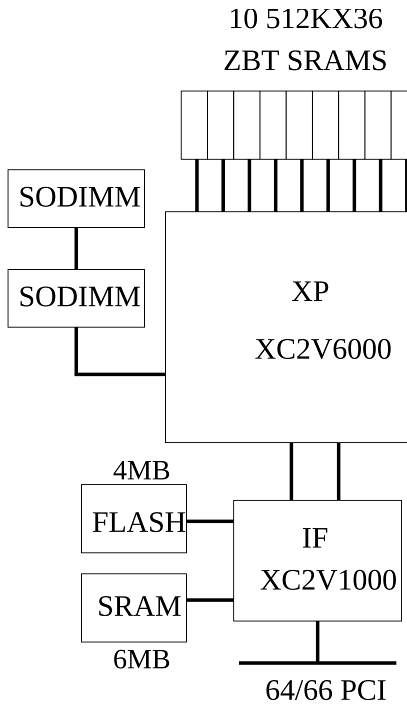


Figure 2.3: Osiris Board Architecture

Two types of memory are also available to the XP. The XP has access to 10 SRAM memories. Each SRAM is a 512Kx36 ZBT capable of running at 200 MHz. The XP can also access two Micron 256MB SDRAMs, for a total of 512MB of PC133 memory. A memory ring component, part of the VHDL library, controls the memory available to the XP. The two types of memory are available to the XP and the host. The host can access the memory using preemption and non-preemption. During preemption, the XP processing will halt while the host accesses the memory. Preemption ensures a seamless visualization by the XP. If non-preemption is preferred, the XP must not access the same memory resource as the host at the same time. The XP can check a memory busy signal to ensure exclusive access to the memory resource [9].

The Osiris board interfaces to the host CPU via a 64-bit 66 MHz PCI connection. The Osiris Board streams data through an input FIFO named FIFO-0 and an output FIFO named FIFO-0. While the two FIFOs have the same name, they are separate FIFOs. Each FIFO is 64 bits wide and 256 words deep. The architecture for the Osiris board is shown in Figure 2.3.

2.2 Smith-Waterman Pattern Matching Algorithm

Pattern matching problems appear in many different disciplines. Algorithms designed to solve the pattern matching problem have evolved over time. This section discusses the Smith-Waterman algorithm [4]. In this algorithm, a pattern P is to be matched against a text T. The pattern P consists of m letters, in other words $P = p_1p_2 \dots p_m$ such that for each i between 1 and m, $p_i \in S$ where S is the alphabet. T consists of n letters of the same alphabet S or, more formally, $T = t_1t_2 \dots t_n$ with each $t_i \in S$. For ease of explanation, the pattern P will always be shorter or equal in length to the text T ($m \leq n$).

Smith and Waterman devised an algorithm for matching similar patterns [4]. The Smith-Waterman (SW) algorithm compares a pattern P to text T and calculates the penalty required to change P into T. Due to the fact that P and T may not match exactly, the penalty will take into account the number of insertions, deletions, and substitutions needed to convert the strings to match each other. This penalty is referred to as the edit distance.

Let s be the penalty for substituting characters if a mismatch is found. Let g be the penalty for a gap. We will assume a gap penalty for an insertion is the same as a gap penalty for a deletion. For instance, if P= CAT and T = CGG, then the penalty would be two substitutions; one for substituting A for G and one for T for G. If P = CAT and T = CAAT then the penalty would be one deletion or gap for removing the second A in sequence T.

$$H_{ij} = max \begin{cases} 0 \\ H_{i-1,j-1} + s(t_i, p_j) \\ max\{H_{i-k,j} - W_k\} \\ max\{H_{i,j-l} - W_l\} \end{cases} \tag{2.1}$$

where $1 \leq i \leq n$ and $1 \leq j \leq m$ and,

H	the SW matrix,
$s(t_i, p_j)$	similarity function between characters, and
W_k	penalty for a deletion of length k.

The SW algorithm solves a similar matching problem using a dynamic programming approach. The complete algorithm is shown in Equation 2.1 [4]. The algorithm uses the solutions from smaller problems to create the larger solution and in the process, creates a matrix of edit distances. The value in cell $H_{i,j}$ represents the degree of similarity between the sequences up to t_i and p_j. A simplification to the SW algorithm was presented by Lipton and Lopresti [11]. In the modified SW algorithm, each element of the matrix uses three other elements to compute its value. Table 2.1 shows the matrix for computing element d.

$$d = min \begin{cases} \begin{cases} a : T_i = P_j \\ a + s : T_i \neq P_j \end{cases} \\ b + g \\ c + g \end{cases} \tag{2.2}$$

			T_i
			$\vdots$
		a	b
P_j	$\dots$	c	d

Table 2.1: Internal Matrix of SW Algorithm

$s = 5$				C	
$g = 8$				$\vdots$	$a = 7$
T_i ='C'			7	12	$b = 12$
P_j ='A'	A	$\dots$	10	d	$c = 10$

$$d = min\{a + s, b + g, c + g\} = 12$$

Table 2.2: Example Matrix Element Calculation

Equation 2.2 shows the modified SW computation required for each element in the edit distance matrix [11]. In this equation, g is the gap penalty, the cost of an insertion or a deletion, and s is the penalty for a substitution, a mismatch between characters. Table 2.2 shows an example computation of a matrix element. Because the characters T_i and P_j result in a substitution, the $a + s$ penalty is used to compute d. The SW Algorithm computes all values in the matrix in order to generate the global edit distance in the bottom right cell. For the n x m matrix, the complexity to compute all values is O$(n * m)$. Because data dependencies exist only on top and to the left of each cell, diagonal values in the SW matrix can be computed concurrently. The parallelized SW Algorithm computes up to n operations at each step. However, due to the data dependency, the algorithm cannot achieve perfect speedup. Thus, the algorithm computes the matrix in $n + m$ steps assuming n available processing elements.

2.3 Gene Matching

The gene-matching problem is actually a set of problems with slight variations. When analyzing a gene-matching algorithm it is important to note whether it is solving global or local alignment, using affine or linear scoring, and if the algorithm compares DNA or proteins [12].

When comparing genetic codes, scientists typically are comparing either deoxyribonucleic acid (DNA) or proteins. DNA is a chemical inside of cells that is made up of four possible nucleotide bases, Adenine, Thymine, Guanine, and Cytosine. These nucleotides are abbreviated A, T, G, and C. The four bases combine in pairs to create rungs in a twisted ladder called a double helix. Adenine always combines with Thymine and Guanine always combines with Cytosine. The bases form words that define the creation of amino acids, the building blocks for proteins. There are 20 different amino acids which chain together to form proteins. Proteins perform a wide variety of activities inside of the body. Missing or altered proteins are the cause of many diseases [1].

Scientists sometimes wish to match entire genes against each other, requiring a global alignment algorithm. However, sometimes, scientists wish only to find a match in a small section of the gene. When only a small section of the gene is required to match, a local alignment algorithm is used [12].

A variation in the way matches are scored provides the last variation in gene-matching algorithms. All of the algorithms have penalties for a gap, an insertion, or a substitution. However, some algorithms provide a larger penalty for an initial gap penalty and smaller penalties for gap extensions. This type of variable scoring uses affine gap penalties. The alternative to affine gap penalties is linear gap penalties, where each insertion or deletion incurs the same cost as the previous one [12].

In addition to the Smith Waterman algorithm, there are a number of other algorithms used to solve the gene-matching problem. Both FASTA [13] and the Basic Local Alignment Search Tool (BLAST) [14] use heuristics to reduce the complexity of the algorithm. Both FASTA and BLAST compare small segments of the query to the database. The assumption in both algorithms is that good matches have a large number of substrings with exact matches. FASTA first computes a position-specific comparison of the genes to generate sets of matches on the same diagonal of the SW matrix. Using the diagonal with the most matches, the SW Matrix is computed for a small band surrounding the chosen diagonal. BLAST operates using a similar technique. However, instead of performing a gapless alignment with a specific gene in the database, the algorithm compares the query sequence against a set of genes with common structure, function or evolutionary origin. BLAST associates a higher scores when the set of genes have common entries. BLAST then takes the highest matches and computes a small section of the SW matrix. These heuristic approaches can miss matches and produce false positives. Both FASTA and BLAST improve the speed of the SW algorithm at the cost of sensitivity in comparing genes. BLAST becomes less precise in finding matches when a family of genes has less in common.

Several companies have produced commercial solutions to the gene matching problem. Time Logic [15] and Compugen [16] have produced products that exploit the use of FPGAs to accelerate their solutions. Both speed up their inner loops of their algorithms by using FPGA boards. The Paracel [17] Gene Matcher2 [18] product uses ASICs and software designed for a Linux distributed system. Implementations of BLAST and FASTA are available for download over the Internet.

2.4 Pattern Matching Applications

Matching patterns is a common function that is useful in a variety of different fields. This section will give an overview of a few of these fields. The pattern matching algorithm discussed in this paper is defined by a large database, a high query load, and noisy data, requiring similarity comparisons.

Network Intrusion Detection (NID) systems are created to use two types of detections. An anomaly detector generates a profile of normal network traffic to be used as the database and queries this database with the current network traffic. A misuse detector queries the network traffic against a preexisting database of attacks, known as signatures. Most intrusion detection systems use perfect match algorithms to compare data. Snort [19], a common packet sniffing program for Linux, uses the perfect matching Boyer-Moore algorithm [20] to compare data.

Fingerprint matching is another important pattern matching application. The first step in fingerprint matching is to identify areas of interest, known as minutiae. The minutiae describe a collection of anomalies within a fingerprint, such as ridge endings or bifurcation [21]. Because fingerprints can be stretched or smeared, most fingerprint applications desire a similarity score rather than a binary match or mismatch. An FPGA implementation of the fingerprint problem was created on the SPLASH-2 [22], an early Xilinx enabled board. The SPLASH-2 implementation has been shown to be over 1,500 times faster than a sequential solution on a SPARC workstation.

Voice recognition and face detection applications also use similar matching algorithms. It has been shown that the face detection problem is a two dimensional representation of a voice recognition problem and is simple to translate between the two applications. The Hidden Markov Model (HMM) is a popular model used in voice [23] and face recognition. The HMM

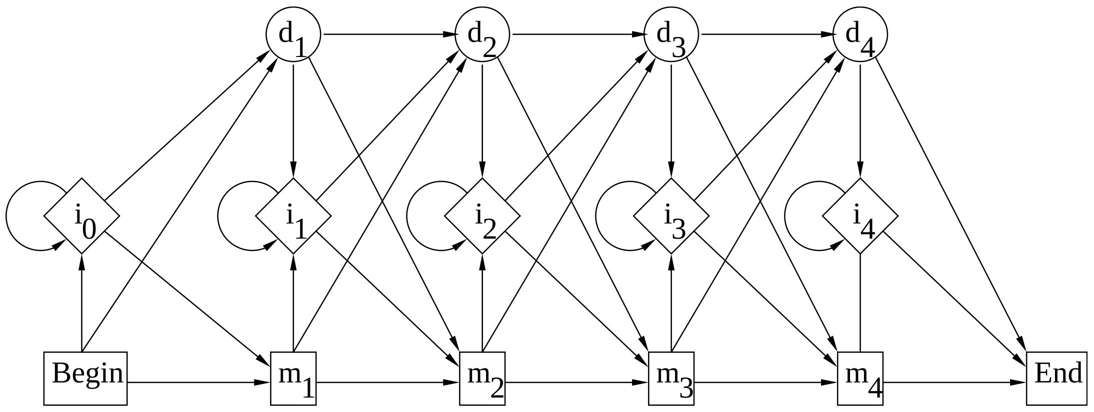


Figure 2.4: Example of a Linear Hidden Markov Model

uses state transitions to represent a sequence alignment. There are three types of states in the HMM, match (m) states, insertion (i) states, and deletion (d) states. Lines between states represent probabilities in transitioning between those states. The HMM model uses position-specific scoring, rather than pairwise scoring as in the Smith-Waterman, BLAST, and FASTA models. The HMM is compared or aligned to a query sequence to determine if the sequence belongs to the same family as the HMM. Figure 2.4 shows a simple HMM model. A parallelized HMM application was implemented to match genes by Hughey [24].

Chapter 3

ACS API Description

The ACS API is an object-oriented approach to controlling distributed adaptive computing systems. The design of the ACS API is scalable and allows users to write single node applications or develop applications using many nodes. The ACS API is a cross-platform interface that currently supports both Windows and Linux. By developing the code in an open-source environment, the ACS API is able to evolve and support more FPGA boards as well as allow users to increase its functionality. A set of default behaviors are defined by the ACS API, which makes the API a simple way to control complex systems. The user need only write one host program to control the entire system regardless of the number of nodes in the system. This chapter provides an overview of the ACS API presented in [26] and [27]. The three sections presented in this chapter divide the ACS API into broad categories of functionality for controlling the distributed ACS environment. The system management section describes the methods for controlling the overall system. The second section discusses how the API controls individual boards. The final section describes the methods for transmitting data to and from the boards in the system.

3.1 System Management

The *host program* is simple program written by the user that controls the adaptive computing system. The first stage of the *host program* is to instantiate a *system* object. The *system* is composed of an array of *nodes* and an array of *channels*. *Nodes* are objects that represent FPGA boards in the system. *Channels* are logical connections between *nodes'* FIFO ports.

The user creates an eXtensible Markup Language (XML) [28] file to define the system. Listing 3.1 shows an example ACS API configuration file. The *commtype* field defines the interprocessor communication type, such as the Message Passing Interface (MPI)[29]. Within the *boards* element, any number of *nodes* may be defined and configured with the *PE* field, such as the Osiris board shown in the example. The array of *channels* is constructed in the *channels* field. Each *channel* element defines the endpoints using the *src* and *dest* attributes. The user's XML configuration file is a simple, reusable method for describing the ACS system.

Listing 3.1: Sample XML Configuration File

```
<?xml version='1.0' encoding='UTF-8'?>
 <!DOCTYPE config SYSTEM "/project/acs_api/src/config/acs_conf.dtd">
<config>
<header>
   <commtype> MPI </commtype>
   <version> 0.03 </version>
   <author> William J. Worek </author>
   <desc> This is a sample config file </desc>
</header>
<boards>
   <osiris board_id="node0" location="local" ctrl_proc_dir ="/project/acs_api/bin" >
```

```
        <PE pe_num="0" prog_file="xp.bit" />
    </osiris>
        ⋮
</boards>
<channels>
    <channel>
        <src> <host port="HOST_OUT"/> </src>
        <dest> <node board_id="node0" port="OSIRIS_FIFO_A0" /> </dest>
    </channel>
        ⋮
</config>
```

The host program uses the *ACS_XMLConfig* class to parse the XML configuration file and create the *system* object. Listing 3.2 contains the beginning code segment for an example ACS host program. After instantiating an *ACS_XMLConfig* object, the XML configuration file name is set. The XML file is parsed during the *getSystem* function to generate an array of nodes and an array of channels. The arrays are then passed to the *ACS_Initialize* function to instantiate the *system* object. *ACS_Initialize* also starts the *control processes* on each host processor in the system.

Listing 3.2: Example ACS Host Program

```
#include "acs.h"
#include "acs_system.h"
#include "acs_xmlconfig.h"
void main(int argc, char ** argv)
{
    ACS_SYSTEM * system;
```

```
int  node_id, node_num;
ACS_XMLConfig config;
config.setFilename(argv[1]);
config.getSystem(&system);
config.getNodeCount(&num_sites);
⋮
```

3.2 Board Management

The ACS API provides several functions for managing individual boards. Boards may either be local on the host machine or remote on a networked machine. The ACS API uses an object-oriented design to hide the location of the nodes in the system. Figure 3.1 shows the class hierarchy implemented in the API. Local nodes implement functions by calling the underlying board-specific API. Remote nodes package the parameters of the function and pass them across the network.

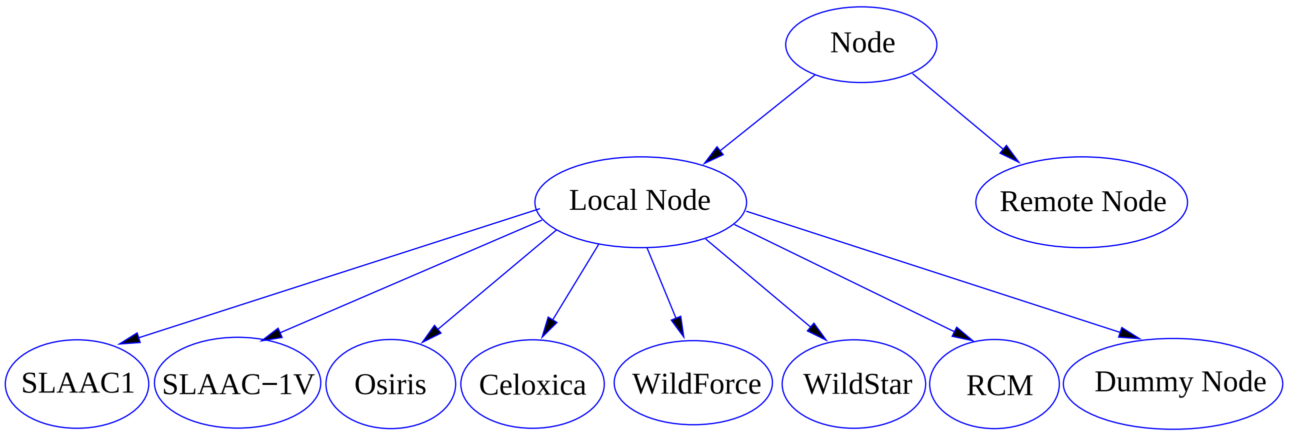


Figure 3.1: Hierarchy of Node Classes

Listing 3.3 shows a code segment from an example host program that demonstrates the use of board management functions. FPGA boards have user clocks that are used to control the speed of the implemented hardware. The *ACS_Clock_Set* and *ACS_Clock_Get* are used to control and query the clocks. The *ACS_Run* and *ACS_Stop* functions are used to turn the clocks on and off. The FPGA has reset signals that are controlled by the *ACS_Reset* function. Finally, there are methods, not shown in the example, for loading hardware configurations onto the FPGA, such as the *ACS_ConfigureFromFile* function. The ACS API reduces information sent to the nodes by caching FPGA configurations on the board.

Listing 3.3: Example ACS Host Program (cont. from Listing 3.2)

```
ACS_CLOCK clock;
for (node_num = 0; node_num < num_sites; node_num++) {
    config.getNodeNumber(node_num,&node_id);
    clock.frequency = 50.0;
    clock.countdown = 0;
    ACS_Clock_Set(&clock, node_id, system, &status);
    ACS_Clock_Get(&cur_clock, node_id, system, &status);
    ACS_Reset(node_id, system, pe_mask, 1, &status);
    ACS_Run(node_id, system, &status);
    ACS_Reset(node_id, system, pe_mask, 0, &status);
}
```

3.3 Data Management

This section describes the functions used for passing data and control information between the host program and the nodes. Specifically, the methods for streaming data using FIFOs, accessing onboard memory, and manipulating registers are described. The ACS API, as mentioned in Section 3.2, makes all board functions uniform despite the location of the node in the system.

Listing 3.4 shows the final code segment of the example host program. This segment of code shows examples of the methods for communicating with the FPGA board. The *ACS_Dequeue* and *ACS_Enqueue* functions stream data through the specified port on the host node. The *control process* passes the data through the ACS *channels* to the board according to the structure defined in the *system* object. Registers can be used for a variety of functions, including sending control information to the FPGA, and providing status information such as the FIFO empty and full registers. The *ACS_Reg_Read* and *ACS_Reg_Write* functions take parameters for the node, address, and size of the register. Onboard memory can be used for providing initialization data or retrieving accumulated data post processing. The *ACS_Write* and *ACS_Read* functions require parameters specifying the node, processing element, and memory bank.

Listing 3.4: Example ACS Host Program (cont. from Listing 3.3)

```
unsigned char     out_buffer [8];
unsigned char     in_buffer [8];
ACS_Dequeue(in_buffer, 8*sendsize, 0, system, &status);
ACS_Enqueue(out_buffer, 8*sendsize, 1, system, &status);
ACS_REGISTER *reg;
reg.size = 64;
ACS_Reg_Read(system, node_id, 0x17, reg);
ACS_Reg_Write(system, node_id, 0x18, reg);
ACS_ADDRESS *addr;
addr.pe = 1;
addr.mem = 4;
```

```
    addr. offset  = 0;
    ACS_Read(out_buffer, 1024, node_id, address, system, &status);
    addr.pe = 2;
    ACS_Write(in_buffer, 1024, node_id, address, system, &status);
    ACS_System_Destroy(system, &status)
    ACS_Finalize();
}
```

Chapter 4

ACS API User Interface

The ACS API was designed to provide a simple, portable, and scalable interface for developing applications on adaptive computing systems. This section describes two interfaces developed as part of this thesis for streamlining the process of creating ACS applications. The ACS API provides a textual interface that allows the user to run scripts and run ACS commands at a command prompt. The second tool is a graphical user interface (GUI), which is the simplest method for interacting with the system.

4.1 Textual Interface

The ACS API textual interface is a useful tool for debugging and constructing applications. The textual interface was developed using the tool command language (TCL) scripting language for creating the textual interface. TCL was developed at the University of California, Berkeley in the 1980s [30]. The scripting language was initially used to run tools for designing integrated circuits.

TCL, like other scripting languages, are useful for "gluing" applications. Scripting languages provide a higher level of programming compared to system programming languages. System programming languages are typically faster and have stronger typing. It is possible to create entire applications simply by creating a script. Applications may also be debugged by running the program command by command.

The ACS textual debugger provides an interface with all the ACS API functions available to the user. The user can combine several function calls into a script. Scripts may be written to perform an entire application and are run by executing them as a command line parameter. Commands can also be run by starting an ACS debugging shell and typing in each command one by one.

While the ACS textual debugger is capable of running applications, it is recommended for debugging purposes only. TCL, like most scripting languages, is an interpreted language. As such, it is unlikely that a script run program will exhibit the same performance as a conventional program created using a system programming language. But, because TCL is interpreted, debugging applications is simpler since scripts do not need to be recompiled every time a change is made.

The textual debugger uses the *tclreadline* package. *Tclreadline* provides the user with a number of convenience functions, such as tab completion and command history. Tab completion is a useful feature that makes learning how to use the ACS API easier because the user no longer needs to remember the full name of the command.

4.2 Graphical Interface

Packaged with the ACS API is a graphical user interface intended for debugging applications. The graphical interface uses Qt [31], a cross-platform C++ GUI development kit. The ACS graphical interface is a simple and effective tool for testing and debugging the functionality of a hardware design without having to create a specific program.

The graphical interface to the ACS accepts an XML configuration file as an optional command line input. Alternatively, the user may configure the system by opening an XML configuration file from the file menu. Once the system is configured, a set of tabs appears in the main window. Each tab represents a different set of functions for accessing the FPGA board. The user may change between boards by selecting the appropriate node from the view menu.

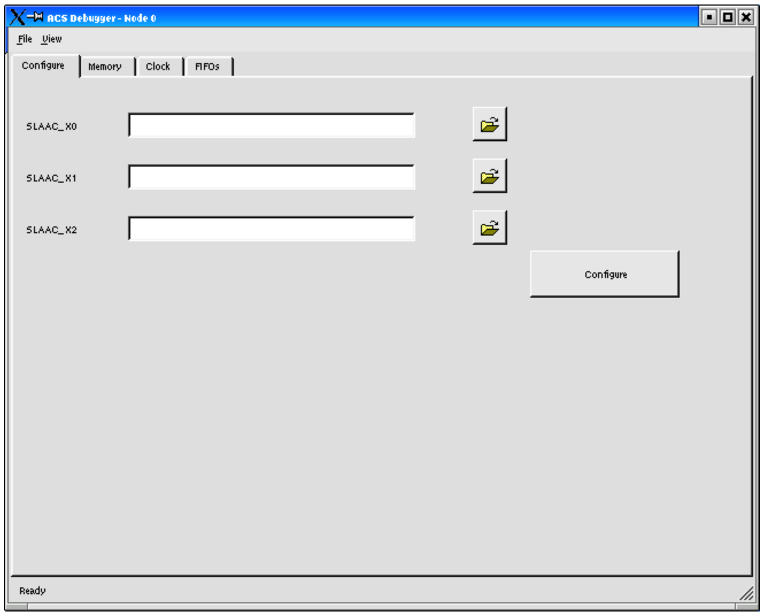
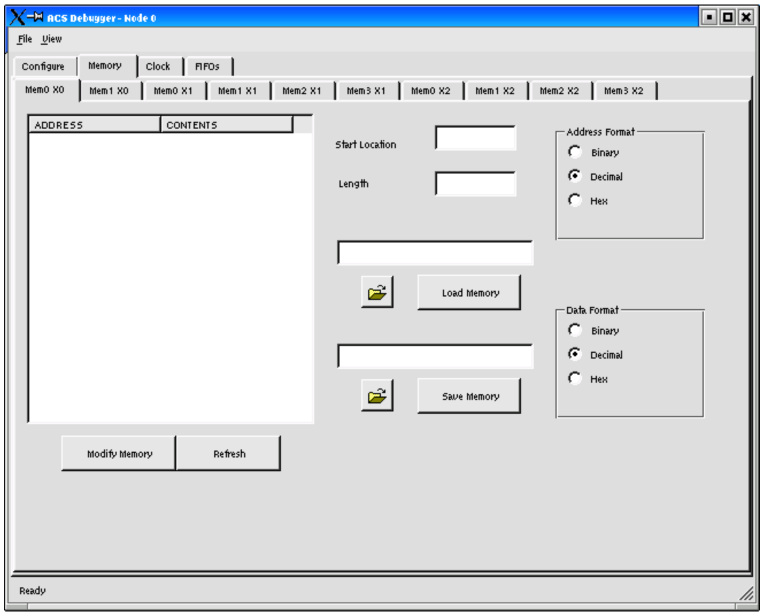


Figure 4.1: Screen Capture of the Configuration Tab

The ACS GUI program generates the appropriate interfaces for the system by querying each board. The node returns information regarding the number and names of devices on the board. For instance, the SLAAC-1V board has three processing elements named X0, X1 and X2. The tab that allows the user to configure the processing elements is able to display the three sets of objects for the node, as shown in Figure 4.1.

The configure tab has a list of processing elements. Each processing element has a text box for entering the configuration file. Near each text box is a button connected to a file dialog. The file dialog is a graphical method for traversing the directory structure and finding the appropriate configuration file. A configuration button, when pressed, signals the system to read the text boxes for the names of the files and configures the appropriate PEs.



Address Format: Binary [✓ Decimal] Hex

Data Format: Binary [✓ Decimal] Hex

Figure 4.2: Screen Capture of the Memory Tab

The memory tab, shown in Figure 4.2, contains a set of tabs, one for each memory on the board. Text boxes are provided for specifying the starting address and the number of locations the user wishes to read or write. The user may either transfer data to/from a table residing in the tab or a file. The data may be represented in binary, decimal, or hexadecimal format by choosing a radio button from the button group.

All values in the memory tab are validated using an ACS_Validator. The ACS_Validator class is derived from the QValidator class. The ACS_Validator reimplements the fixup and validate functions. The validate function is called every time a key is pressed within the field. This function can limit which keystrokes modify the text field. For instance, if the memory is in a binary form, the only valid keystrokes are 0's and 1's. The fixup command is called when focus leaves the text field, implying that the text is in its final form. Fixup modifies the value in the text field in order to make it valid. For instance, if the memory is 16-bits wide and 17 bits of data are entered into the field, the fixup function can modify it to a valid 16-bit value, either by cropping the last digit or by saturating it to the maximum value.

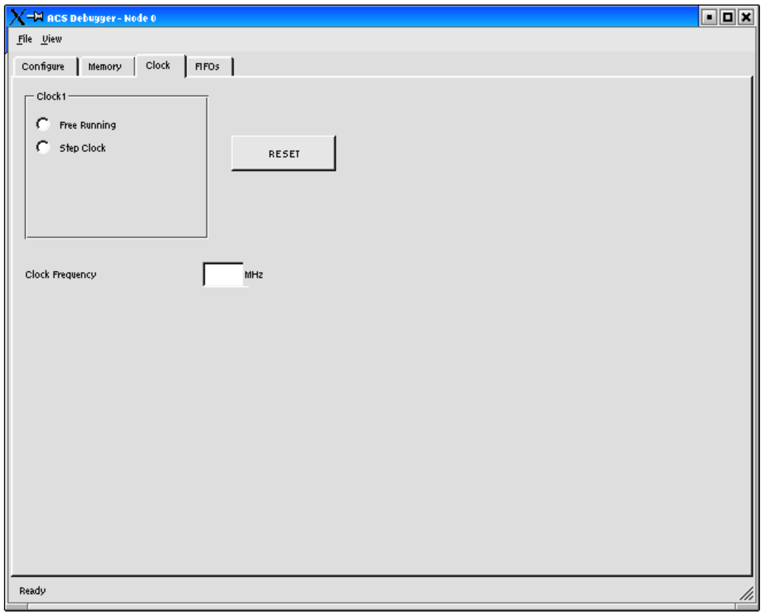
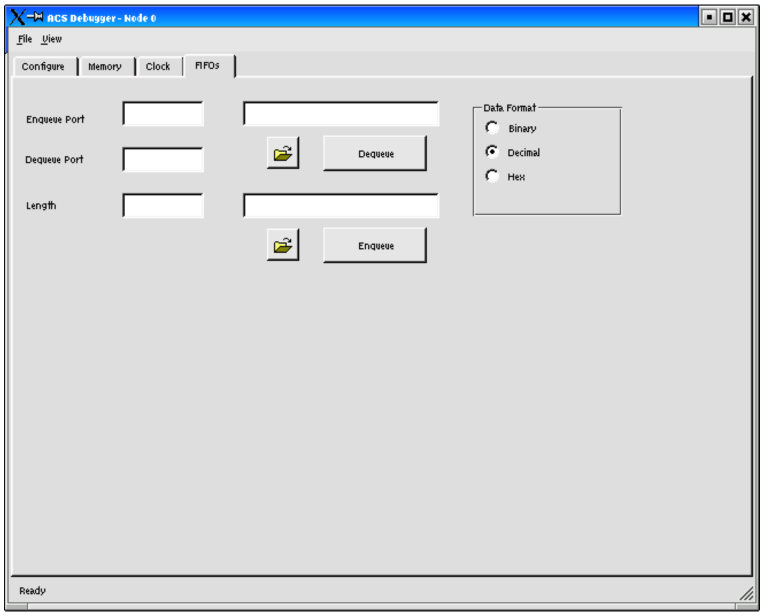


Figure 4.3: Screen Capture of the Clock Tab

The clock tab, shown in Figure 4.3 contains objects for modifying and controlling the clocks on the FPGA. The clock may be engaged in free-running or stepping mode. If the clock is in free-running mode, a toggle button allows the user to start/stop the clock. If the clock is in stepping mode, the button transforms into a push button that steps the clock each time the button is pressed. The clock tab also contains a button for reseting the board and a text field for modifying the clock frequency.



Data Format: Binary [✓ Decimal] Hex

Figure 4.4: Screen Capture of the FIFO Tab

The FIFO tab, shown in Figure 4.4, is used to interface with the streaming data functions of the FPGA board. In the FIFO tab, the user can specify the ports for enqueueing and dequeueing in text boxes. The data for the stream is loaded or saved in a file given by the user in a text field or file dialog. Enqueue and dequeue buttons are used to commit parameters and call the underlying ACS API functions.

The objects in the ACS graphical interface are seamlessly integrated by the Qt tool kit. Qt uses signals and slots to pass information within the program. A function can emit a signal that will traverse the class hierarchy and be caught by all slots that use the signal. Qt programs are first preprocessed to find all the signals and slots using a meta object compiler. The final application results from compiling all of the meta objects together with the original code. Signals and slots are used in the ACS GUI for such things as sending signals to the status bar in order to display information about the success of ACS API functions.

The ACS graphical interface is a powerful tool in debugging adaptive computing systems. The user can control all functions of the board without learning any of the function calls of the ACS API. While controlling FPGA boards using the ACS graphical interface is slower than that of a program created using C or C++, and even slower than using the textual interface, using the ACS GUI allows users to control distributed adaptive computing systems without any knowledge of the underlying APIs.

Chapter 5

FPGA Pattern Matching Implementation

The following section describes the runtime reconfigurable hardware implementation of the Smith-Waterman [4] genetic code matching algorithm that was adapted from [32]. The first section gives a top-level overview of the hardware design. The following sections describe the parts within the design, namely the processing element, the state machine, the up/down counter, and the data control.

5.1 Overview

The FPGA implementation of the Smith-Waterman algorithm consists of four parts, the processing elements, the state machine, the up/down counter, and data control. The processing element computes the matrix value for each character in the query string. A state machine follows the processing element to convert the output of the processing elements into

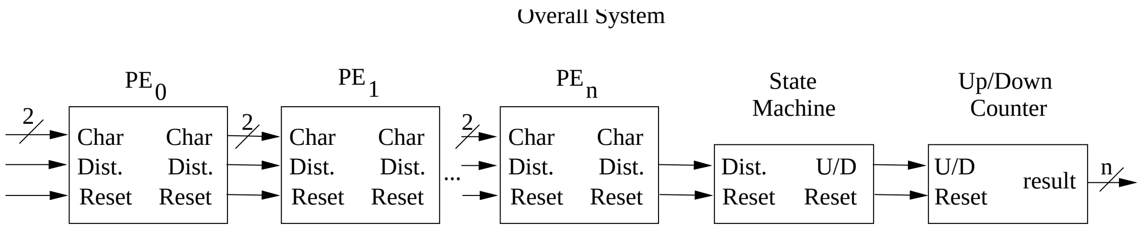


Figure 5.1: Block Diagram of the FPGA Implementation

an up/down signal. The up/down signal connects to an up/down counter that computes the final edit distance value. The method for receiving and sending data through the system is the final part of the implementation. Figure 5.1, adapted from [32], shows the overall system of the implementation.

An important objective in the design for the hardware was to maximize the number of processing elements that fit on an FPGA. If a query sequence cannot fit on one board, it must span either boards or configurations. Nucleotides are represented by a two-bit value as shown in Table 5.1. Insertions or deletions have a penalty of one and substitutions have a penalty of two. Using these penalty values reduces the amount of data that needs to be stored in the PE and sent to the next PE. Because the insertion and deletion penalty is one, the value in the matrix cell a differs from values in cells b and c by exactly one. The d cell differs from the a cell by either zero or two.

A	T	G	C
00	01	10	11

Table 5.1: 2-bit Nucleotide Representation

Intermediate edit distance values are computed and stored in each PE every clock cycle. Because adjacent cells vary by exactly one, the least significant bit can be inferred. The intermediate edit distances are computed modulo four, reducing storage space required for the matrix value in the processing elements. Because successive edit distance values vary by exactly one, no information is lost if the values are stored modulo four.

5.2 PE Description

The hardware implementation of the Smith-Waterman algorithm has processing elements that compute the values in each column of the edit distance matrix. The query character that is used to compute the column is folded into the hardware logic. Each gene matching PE fits within three slices, or less than one Virtex II CLB.

This implementation of the sequence comparison algorithm uses only local signals. Each processing element requires only the edit distance from the previous column. All of the components in this design are reset synchronously. The database sequence is streamed through the PE, resulting in a single matrix computation each clock cycle per PE.

The block diagram of the inside of a processing element is shown in Figure 5.2, adapted from [32]. The boxes on the right represent flip-flops. The boxes on the left represent LUTs and inside each LUT is the function the LUT generates. The synchronous reset initializes the flip-flops to appropriate values for each processing element. As seen from this diagram, the processing element fits within three slices. The fourth slice of the CLB is left empty for future improvements.

5.3 State Machine

The final processing element sends its local edit distance value into a state machine. The state machine, in combination with the up/down counter, is responsible for converting the

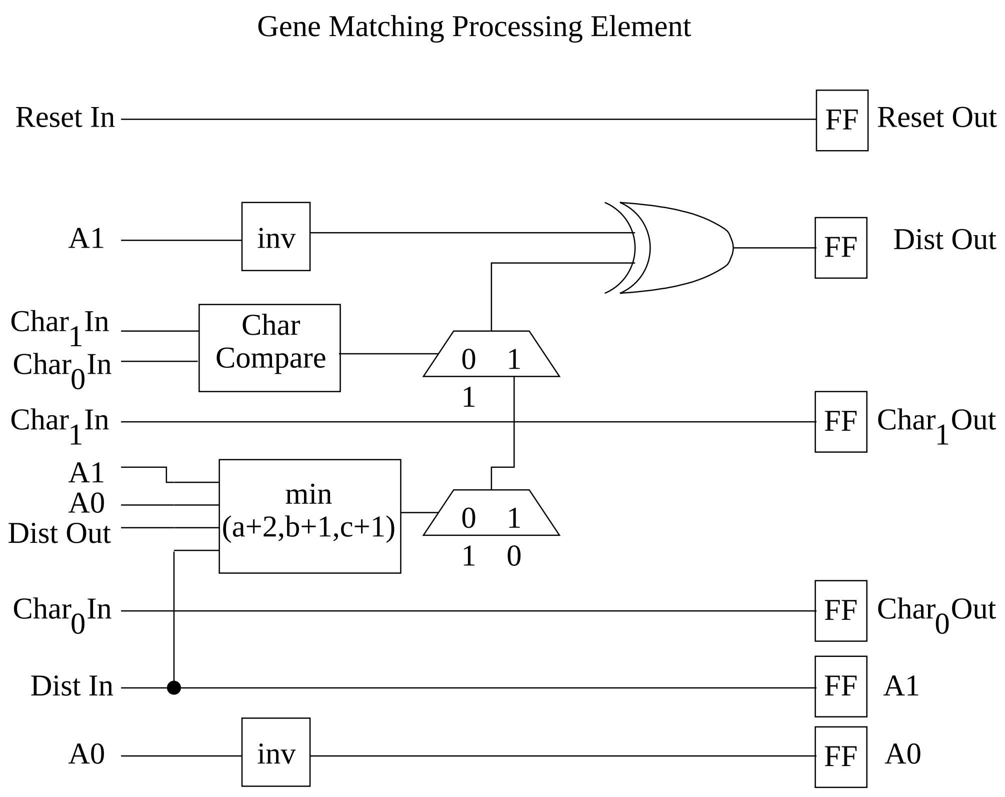


Figure 5.2: Diagram of the Inside of a Processing Element

edit distance from a two-bit, modulo four representation into the full edit distance value. Figure 5.3 shows the state machine used for this gene matching implementation.

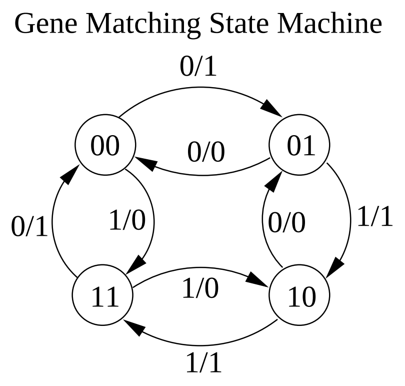


Figure 5.3: Gene Matching State Machine

As shown in Figure 5.3, the state machine has four states. Each state represents the four possible intermediate edit distance values. The initial state depends on the size of the query sequence. The edit distance from the processing elements determines the next state of the system. Only the upper bit of the edit distance is required because the lower bit is inferred. The outer loop in the figure sends an up signal while the inside loop sends the down signal. The up/down signal that is generated is passed to the next stage, the counter.

5.4 Up/Down Counter

At the end of the circuit is an up/down counter that calculates the final edit distance value. The counter receives an up/down signal from the state machine and outputs the 16-bit edit distance and four reset signals. The up/down counter is composed of four 4-bit pipelined counters. The reset signals that are output represent the reset for each of the four stages

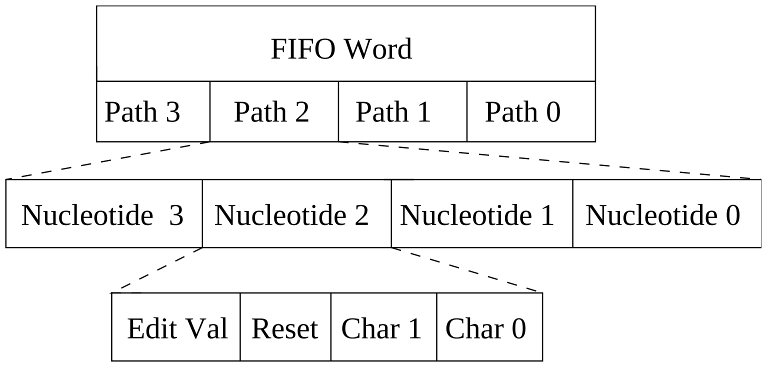


Figure 5.4: FIFO Word

of the pipeline. In order to view the final edit distance value, the signals output from the counter must be captured in four successive clock cycles, starting with the least significant bits.

5.5 Data Control

In order to use the resources of the FPGA efficiently, knowledge of the speed of the devices was required. Any communication between the FPGA board and the host must travel through the PCI bus. An analysis of the design shows that the goal of making the user clock on the FPGA run approximately four times faster than the PCI bus was met. The 64-bit FIFO word, shown in Figure 5.4, is divided into four parts, where each 16-bit part is read into the circuit each clock cycle. The 16-bit part is further divided into four 4-bit nucleotides. Each nucleotide is input into four paths that reside on the FPGA.

The clock used by the system is gated by the FIFO input empty flag, representing no valid data available, and the FIFO output full flag, representing no available space to pass data. FIFO words are written to and read from the host every fourth clock cycle. Due to the four to one ratio in clock frequencies, the algorithm cannot compute final edit distance values for genes with less than four nucleotides. If a query sequence cannot fit on one FPGA, an intermediate configuration is required. Intermediate configuration files reconstruct the FIFO word in the same fashion as the FIFO word that was input to the board. Final configuration files use all 16 bits dedicated to their path to output the 16-bit edit distance after each database sequence is completely passed through the system.

A second method for transferring the database to the board was implemented using the FPGA board's memories. The host writes to the memories with the database to be streamed through the system and triggers the FPGA by enqueueing a single value to the board. The FPGA then progresses out of its waiting state and streams the data from the memories and through the processing elements. The resulting edit distance values are then written back into memory. After streaming the entire contents of memory, the FPGA enqueues a single value letting the host know that the memories are ready to be read.

Chapter 6

ACS System Implementation

6.1 Overview

One problem that arises in the the solution to the gene-matching problem is that the query sequence embedded in the FPGA design may not fit within a single FPGA configuration. Two solutions to the problem are presented in this chapter. The first system solves the problem by spreading the search sequence across multiple boards and streaming the database sequentially through the boards. The second system uses a single board for each search sequence and reconfigures the board after passing the database through each time. This system scales by distributing the database search over multiple boards.

Before the system can be used, the configuration files must be generated. The hardware described in Chapter 5 embeds the query sequence into the hardware configuration. Each new sequence requires a new configuration to be generated. Figure 6.1 shows the flow of creating new logic core configurations given the query gene sequence and an ASCII representation of the generic configuration design in the Xilinx Design Language (XDL). The query sequence

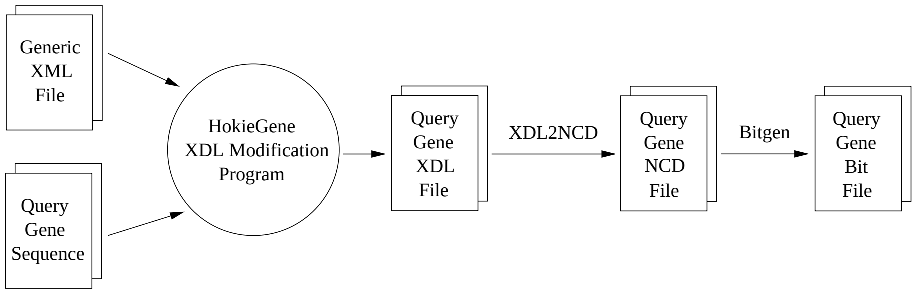


Figure 6.1: Customized Core Generation Flow

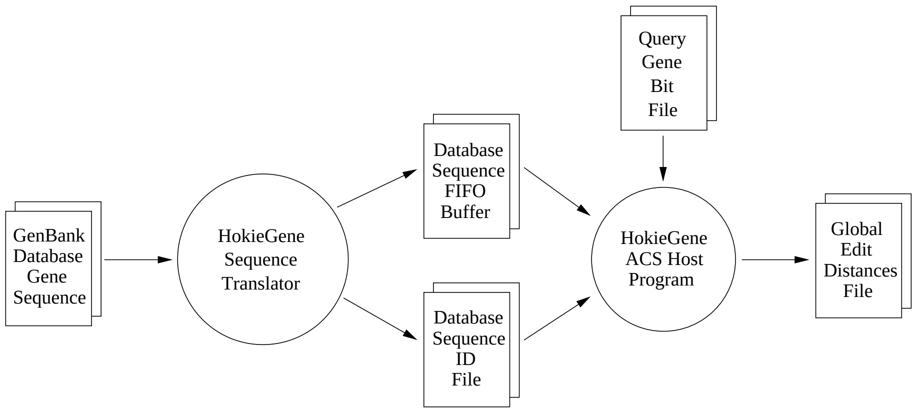

GenBank Database Gene Sequence
HokieGene Sequence Translator
Database Sequence FIFO Buffer
Database Sequence ID File
Query Gene Bit File
HokieGene ACS Host Program
Global Edit Distances File

Figure 6.2: Program Flow

XDL file is translated into the native description format (NCD) and then completes the regular Xilinx flow for bitfile generation.

In addition to generating FPGA configuration bit files using the query sequence, the gene sequence database must also be translated into a usable format for the ACS host program. Figure 6.2 shows the flow of the gene sequences in the overall software system. Gene sequence databases can be obtained from one of several gene repositories. The nucleotide sequences used in this thesis were obtained from GenBank [2]. The GenBank representation of the genes is first transformed into the FIFO word format, as shown in in Figure 5.4. This

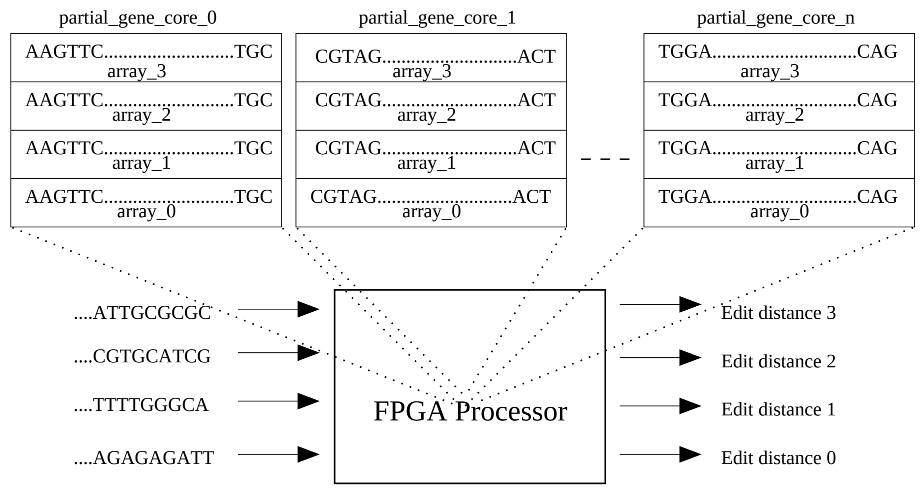


Figure 6.3: Architecture of a Node in the Streaming System

transformation removes all of the gene information that is stored with the sequence in the database. A second file that contains the identification values for each sequence is also generated. These files, along with the query sequence configurations, are input to the ACS host program and the final edit distances of the SW algorithm are computed.

6.2 Streamed System

The streamed system solution spreads a single genetic sequence across multiple boards. The database is then passed through each FPGA board in the system. The final edit distance value is read from the final FPGA board. Figure 6.3 shows a diagram of the distribution of query sequence configurations across the FPGAs in the system.

Once the data is formatted properly, the nucleotide sequences are enqueued into the system through port 0. From the host program, the data is transferred into the *dummy node*. The control process thread on the host machine dequeues the data from the dummy node and

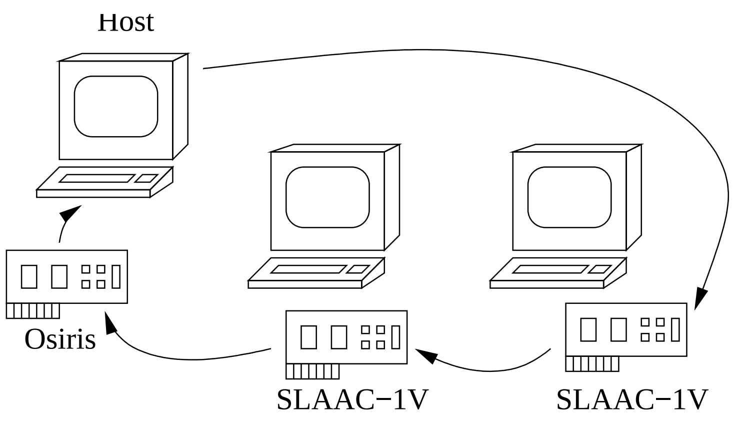


Figure 6.4: Streaming System

places it into the channel buffer. The control process periodically attempts to flush the data in the channel buffer by enqueuing the data into the first node.

The first node processes the data and outputs an intermediate data stream. The data is dequeued from the node by the control process and sent to the next node. This process continues until the data finishes traversing all of the nodes. At the final node, only the edit distance for the gene sequence is output. The final channel dequeues the edit distance and passes it back to the dummy node. The host program can then dequeue the edit distances from port 1 and present the data to the user.

Figure 6.4 shows the three-node implementation of the streaming system. The host machine contains an Osiris Board with the final configuration. The two other computers are accelerated with SLAAC-1V boards. Separate implementations using two and three nodes were developed and tested. The results from the systems are shown in Chapter 7.

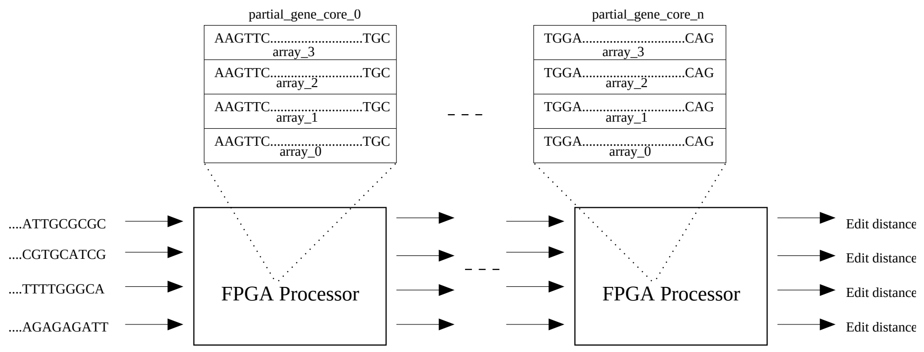


Figure 6.5: Architecture of the Distributed System

6.3 Distributed System

The distributed system solution makes use of multiple boards by streaming data through them independently. In the distributed system, the genetic search sequences are spread across configurations instead of FPGA boards. Depending on the length of the query sequence to be searched, a single FPGA board may need to be reconfigured multiple times. Figure 6.5 shows the distribution of configuration files in a single FPGA for the distributed system.

Once the database of genetic sequences is formatted properly as mentioned in Section 6.2, the host program broadcasts the database sequence to every node in the system. Because each node has an intermediate configuration file, the host program must dequeue and store the intermediate values. Once the entire database has been streamed through the system, the nodes can be configured with the next query sequence core.

The process of streaming data through the nodes, storing the intermediate values, and reconfiguring the board must be repeated until the search sequence is exhausted. When the final configuration file is loaded, the host process dequeues only the final edit distance values and then presents the data to the user.

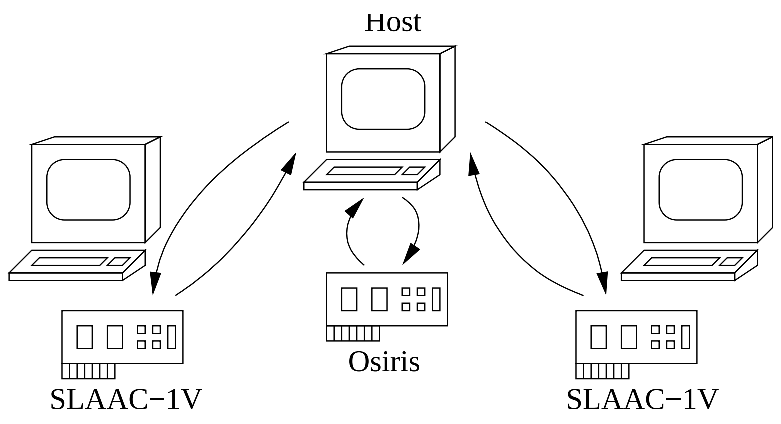


Figure 6.6: Distributed System

Because the host machine can become a bottleneck in the system, the fastest computer was selected for running the host program. The host computer, accelerated with an Osiris board, is networked to two remote machines, each outfitted with a SLAAC-1V. Figure 6.6 shows a diagram of the distributed system.

Chapter 7

Analysis and Results

This chapter presents the results for the genetic sequence matching implementation using heterogeneous distributed adaptive computing systems. The first section compares the results of using the ACS debugger compared to an ACS host program and discusses the use of the debugger in developing an ACS application. The second section discusses the area requirements for the FPGA implementation and the benefits of using a run-time reconfigurable approach. The following section discusses the theoretical results of this implementation and compares those results with other commercial and academic sequence matching systems. The next section analyzes the results of a single node application and compares using local versus using remote nodes and using the ACS API versus using the native SLAAC API. The final two sections show results for two heterogeneous distributed adaptive computing systems and describes the tradeoffs in each system's design.

7.1 ACS Debugger

The ACS textual debugger allows the user to run and debug applications using commands at a prompt or by executing a script. The textual debugger was created using an interpreted language, TCL. This section compares using the textual debugger versus using a standard C++ system program. While the debugger is a useful tool for stepping through applications, it lacks the performance required to make it an efficient method of running a final design.

Table 7.1 shows a comparison of the results using the textual debugger for the SLAAC-1V and the Osiris boards. The entries in the row marked *Configuration Time* represent the time in seconds that it takes to configure all three PEs on the SLAAC-1V or the XP user FPGA in the case of the Osiris. The memory transfer rates show the achieved throughput of a full memory write followed by a full memory read of a single ZBT RAM on each of the boards. The FIFO transfer rates reflect the throughput of a pass-through FIFO in each of the FPGA boards.

	SLAAC-1V		Osiris	
	Host Program	Textual Interface	Host Program	Textual Interface
Configuration Time	1.52s	1.634s	.610s	.670s
Memory Transfer Rate	6.58MB/s	.288MB/s	72MB/s	1.75MB/s
FIFO Transfer Rate	.935MB/s	.036MB/s	111MB/s	.923MB/s

Table 7.1: Results for the Textual Interface

The textual debugger's performance is significantly less than that of an ACS host program for two major reasons. First, the textual debugger is an interpreted language and requires that each command be translated into the native machine language. The second reason for the resulting performance deficiencies is due to the overhead in setting up each function call.

For instance, for each enqueue call, a buffer to hold the data must be created and loaded from the argument string and all of the parameters must be converted from to the appropriate data format. But regardless of the performance results, the textual debugger finds its niche by being able to step through commands without any recompiling. The result is a simple to use tool for debugging ACS applications.

7.2 Area

This section analyzes the area requirements for the configuration described in Chapter 5. Included in this section is a theoretical discussion of the potential area usage on the SLAAC-1V and Osiris boards, as well as an analysis of the configurations developed during the testing of the gene matching implementation. The area limitation of the FPGA is an important metric when considering gene matching. Because genetic sequences may contain a large number of nucleotides, one goal of the hardware design is to maximize the number of nucleotides that can be embedded in a configuration, thus reducing the number of boards/cofigurations a query sequence must span.

The state machine, up/down counter, and data management circuitry use a negligible amount of area on the FPGA. The combination of these three parts of the design requires less than 100 Slices or .3% of a Virtex II 6000 device. The bulk of the configuration is composed of the array of gene processing elements. The gene matching processing elements, therefore, are the portions of interest when considering area.

Each processing element in the hardware configuration represents a single nucleotide in the query sequence. Because the nucleotide is folded into the circuitry, the size of each processing element is very small; each processing element requires only three Virtex slices. Because the value of the nucleotide is embedded into the circuitry, the FPGA must be reconfigured for each new query. The overhead of reconfiguration is traded for the ability to fit many processing elements in a single configuration. A non-reconfigurable implementation would require five Virtex slices, effectively reducing the number of processing elements on an FPGA by three fifths.

The SLAAC-1V has three Virtex 1000 parts that each have 6,144 CLBs. Each Virtex CLB contains two slices. Because each nucleotide in the query sequence, translated into hardware, requires three slices, the X1 and X2 PE can fit approximately four thousand query nucleotides. The X0 PE can fit approximately half as many query nucleotides on the device because the X0 PE already contains the interface circuitry for the FPGA board. By streaming data through all three processing elements, a SLAAC-1V has the ability to compute the edit distance of a 10,000 nucleotide sequence without reconfiguration. An implemented X1 design was developed with four paths of 700 nucleotides. After mapping the configuration to the Virtex chip, the entire design required 10,416 slices, or 84% of the X1 device.

The Osiris board has one user FPGA, a Virtex II 6000 part. The user FPGA device has 8448 CLBs. Each Virtex II CLB contains four slices. Therefore, the Osiris board can have a configuration of approximately 7000 gene-matching processing elements. The implementation synthesized for this thesis has four paths each with 1700 processing elements for a total of 6,800 nucleotides. The mapped configuration resulted in 87% of the board, 29,338 slices, being used.

7.3 FPGA Throughput

The primary goal of the implementation was to reduce the time required to compare genetic sequences using the Smith-Waterman algorithm. This was accomplished by maximizing the number of query sequences on an FPGA, and by maximizing the throughput of the system. These two metrics are combined to create the standard commercial metric of Smith-Waterman cell updates per second.

One reason the Smith-Waterman algorithm was chosen to be implemented on FPGA technology was that the SW algorithm uses only local signals. Because all signals are local, the FPGA board can be run at extremely high frequencies. During development, it became clear that the PCI bridge would be the bottleneck in determining the maximum frequency of the circuit.

After synthesizing the Osiris implementation, the circuit was determined, by the synthesizing software, to perform correctly when the user clock is less than 180 MHz. The I/O clock on the board is limited to 45MHz. Because each FIFO word has four paths of four characters, the circuit efficiently uses all of the PCI bandwidth. Theoretically, if the FIFO is kept full, the FPGA circuit can achieve 1.26 trillion cell updates per second.

The SLAAC-1V implementation uses a 32-bit 33MHz PCI bus. By following the same logic of the Osiris implementation, the SLAAC-1V implementation is capable of performing at approximately one quarter the speed of the Osiris board. The circuit was designed such that the user clock could run at 66 MHz. If the FIFO is kept full, the SLAAC-1V board can achieve 462 billion SW matrix updates per second.

Table 7.2 compares the theoretical results of the design presented in this thesis to previous implementations. Both Paracel and TimeLogic are commercial products. Paracel accelerates their system using ASICs, while the remaining designs use FPGAs. Splash-2 is an early FPGA board that was used to implement a one time reconfigurable solution to the gene matching problem. The use of runtime reconfiguration for optimization of the processing elements greatly increases the number of gene processing elements that fit on a device. However, at this time, it takes approximately nine minutes to convert an XDL file into a bitfile configuration using the Xilinx foundation tools. This time is quite large considering the process is simply converting the file from one format to another without requiring any routing or placing. The bitfile creation can be performed on multiple machines to ensure that FPGA boards do not need to wait for the configurations.

	Year	Processors per Device	Devices per Node	Updates per second
Pentium III - 1.4GHz	2002	1	1	82M
Paracel(ASIC)	2001	192	144	276B
TimeLogic (FPGA)	2000	6	160	25B
Splash 2 (XC4010)	1993	14	272	43B
SLAAC1V (XCV1000)	2002	2800	2.5	462B[1]
Osiris (XC2V6000)	2002	7000	1	1.260T[1]
SLAAC1V(acheived)	2002	2800	2.5	23B
Osiris(acheived)	2002	7000	1	389B

Table 7.2: Performance and Hardware Size for Various Systems

7.4 Single Node Throughput

This section discusses the performance of the gene matching system using a single node. The Osiris and SLAAC-1V single node designs were implemented using the SLAAC API, the native API designed for the board, and the ACS API. The ACS API performance results are shown for the single local node as well as the single remote node system. Results are shown for two methods of transferring information to the board, through FIFOs and through memory transfers.

[1]Values represent theoretical results computed using maximum achievable clock speed and area.

$$t_{\mathrm{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * max(t_{clk}, t_{fifo}(D)) \tag{7.1}$$

$$t_{\mathrm{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * (t_{clk} + t_{mem}(D)) \tag{7.2}$$

$$t_{\mathrm{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * max(t_{clk}, t_{fifo}(D), \psi * t_{net}(P)) \tag{7.3}$$

$$t_{\mathrm{db}} = \left\lceil \frac{n}{N} \right\rceil * t_{config} + (n+m) * (t_{clk} + t_{mem}(D) + \psi * t_{net}(P)) \tag{7.4}$$

where,

t_{db}	time to compute edit distance of a data base,
t_{config}	configuration time,
t_{clk}	FPGA clk period,
t_{fifo}	time per nucleotide to stream D bytes block of data to/from the FPGA,
t_{mem}	time per nucleotide to send D bytes block of data to/from the FPGA,
t_{net}	round-trip time per nucleotide of a P-sized packet across the network ,
ψ	remote function overhead factor,
N	maximum number of nucleotides per FPGA,
P	MPI packet size transferred across the network,
D	block size of data transferred to the board,
n	size of the query sequence, and
m	size of the database sequences.

The time required to compare a query sequence to a database of sequences in a single local node is calculated by Equation 7.1 and Equation 7.2. When remote nodes are used, a network communication factor is added, as shown in Equation 7.3 and Equation 7.4. The network communication term is multiplied by a ψ factor in order to compensate for the overhead in packing and unpacking parameters for the remote function calls. If FIFOs are used to transfer database sequences to the board, data transfer can be overlapped with the FPGA computation. In contrast, the memory transfer approach does not overlap computation and I/O. The time to load a configuration on a remote system is approximately equal to the time it takes to reconfigure on a local node because only a small amount of data, the configuration file names, must be passed across the network. Each of the remaining terms are multiplied by the size of the database sequence plus the size of the query sequence.

	SLAAC-1V	Osiris
t_{config}	1.52s	.610s
t_{clk}	1.52e-8s/B	5e-9s/B
t_{fifo}	1.07e-6s/B	9.01e-9s/B
t_{mem}	1.53e-7s/B	1.48e-8s/B
$t_{\text{net}}(fifo)$	1.82e-7s/B	2e-6s/B
$t_{\text{net}}(memory)$	1.82e-7s/B	1.82e-7s/B
$\psi(fifo)$	2.20	1.03
$\psi(memory)$	1.10	3.83

Table 7.3: Values for the Single Node Model

Tablel 7.3 shows the values obtained for the single node models. The configuration for each board was timed to produce the configuration time. The clock time is the speed of the user FPGA clock set by the host program. By running a pass-through configuration on the FPGA, the time to stream data through the FIFOs was found. The memory time was obtained by sending a block of data the size of the ZBT memory to the board and reading it back. The network time was obtained by running an MPI program that sends and receives data from a single remote node. The program calculates the network throughput for various sized packets. The values shown in the table represent the network throughput for sending memory-sized packets and FIFO-sized packets. The ψ factor is computed using the time required to stream data and read/write memory on a remote node.

The results for this thesis were obtained using an Osiris board in a machine that contains a Pentium III 1.4 GHz processor with 512MB of RAM. The results for the single node SLAAC-1V implementation were shown using a Pentium II 300MHz machine with 256MB of RAM. A second SLAAC-1V, used in the multiple node systems, is in a Pentium II 333MHz machine with 384MB of RAM. Each SLAAC-1V accelerated machine is networked together by a 10/100 switched ethernet connection and the Osiris board is connected through a gigabit network.

		SLAAC-1V			Osiris		
	Database	SLAAC	ACS		SLAAC	ACS	
	Size	Local	Local	Remote	Local	Local	Remote
GBUNA	.159MB	.115s	.183s	.743s	.002s	.002s	.0319s
GBPHG	3.27MB	2.34s	3.75s	15.0s	.030s	.031s	.279s
GB14	11.2MB	7.96s	12.7s	51.2s	.103s	.101s	.936s
GBGSS4	19.2MB	13.9s	21.4s	88.1s	.177s	.173s	1.60s
GB36	28.8MB	20.4s	31.7s	133s	.267s	.259s	2.41s
GB48	38.4MB	27.3s	41.1s	178s	.355s	.345s	3.22s
GBPRI20	47.5MB	33.7s	50.9s	219s	.438s	.425s	3.97s
Throughput		1.4MB/s	.94MB/s	.22MB/s	108MB/s	111MB/s	12MB/s

Table 7.4: FIFO Results for the Single Node

The ACS API and SLAAC API host programs are very similar. The main differences lie in the construction and initialization of the systems. The exact same software algorithm is used in each version of the program resulting in a good direct comparison of the results. Table 7.4 shows the results when using the FIFOs to transmit data to the board. The sample query gene was compared to four different databases of genBank genes, GBUNA, GBPHG, GBGSS4, and GBPRI20. In addition to the genBank genes, three additional databases GB14, GB36, and GB48 were used. These three gene databases are subsets of the GBPRI20 database created to show additional points of comparison. The size of the database sequences after the transformation into FIFO/Memory word format is shown in Table 7.4. The system throughput represents the slope of lines shown in Figure 7.1 and Figure 7.2. The system throughput of the Osiris board is limited by the PCI bus speed in the local case and limited by the network speed in the remote case. Baseline tests for the network were experimentally shown to be approximately 6MB/s for the SLAAC-1V and 15MB/s for the Osiris in the best case (transferring large packets of data) and only about 1MB/s in the worst case (small packets of data). These values were obtained by using a MPI program that varies the send/receive sizes for node to node communication. Because the SLAAC-1V driver only allows 250 word FIFO transfers, the system becomes inefficient due to the overhead of setting up each transfer.

Figure 7.3 and Figure 7.4 shows the time versus database size for the SLAAC-1V. The slope of the lines represent the system throughput and the y-intercept represents the time required to configure the boards. The graphs show the time difference of running the system using the ACS API and SLAAC API. The theoretical results, calculated using the local and remote node models, are also shown.

Table 7.5 shows the values for using memory to transfer data to the board. The memory transfer sizes are limited by the size of the memories on the board. The SLAAC-1V can transfer 256K words to the board for each iteration. The Osiris board has 512K word memories and both boards memories are effectively 32 bits wide. The performance results for the SLAAC-1V are better than the FIFO implementation because the transfer sizes are

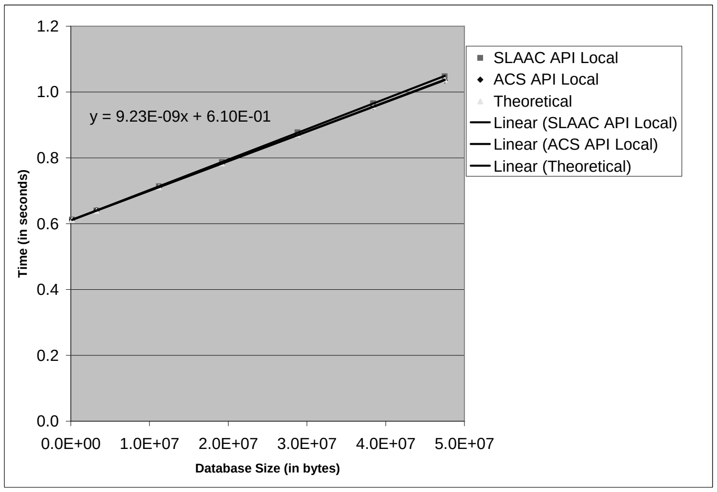


Figure 7.1: Single Local Osiris Board using FIFOs

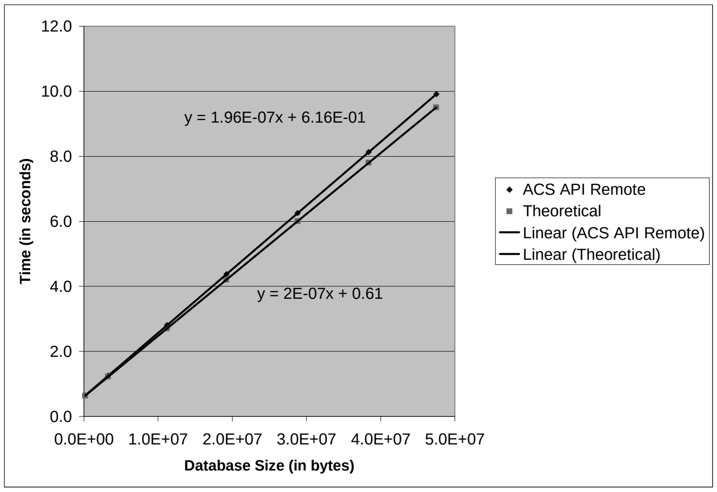


Figure 7.2: Single Remote Osiris Board using FIFOs

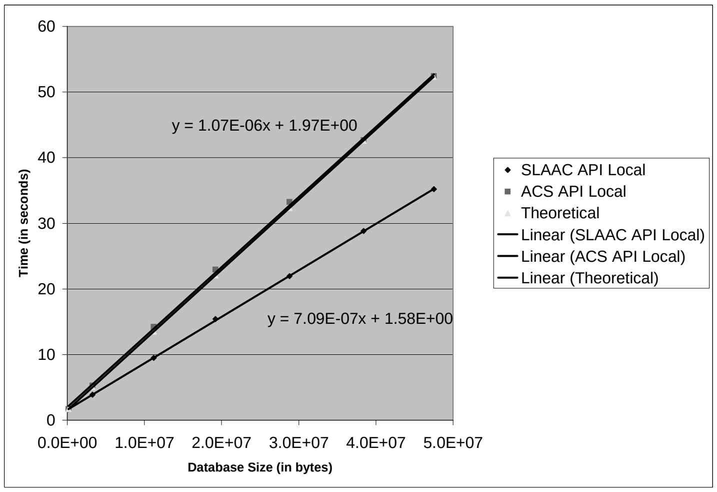


Figure 7.3: Single Local SLAAC-1V Board using FIFOs

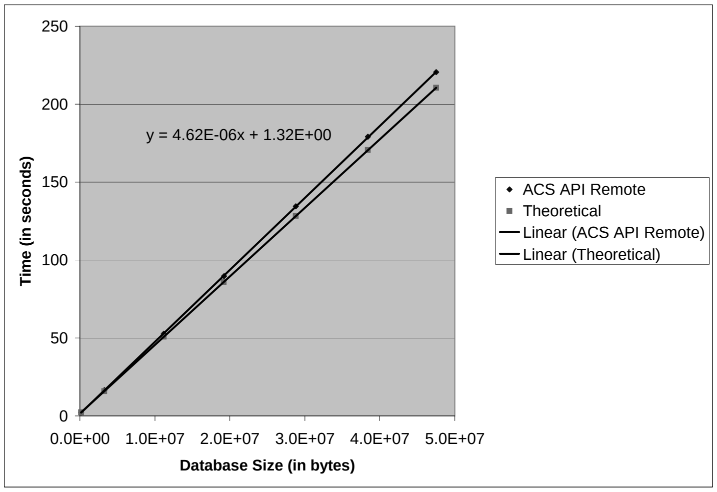


Figure 7.4: Single Remote SLAAC-1V Board using FIFOs

Board		SLAAC-1V			Osiris		
API	Database	SLAAC	ACS		SLAAC	ACS	
	Size	Local	Local	Remote	Local	Local	Remote
GBUNA	.159MB	.122s	.129s	.347s	.026s	.004s	.211s
GBPHG	3.27MB	.611s	.648s	1.72s	.051s	.060s	3.11s
GB14	11.2MB	3.06s	1.94s	5.14s	.153s	.200s	10.4s
GBGSS4	19.2MB	7.34s	3.24s	8.62s	.254s	.336s	17.7s
GB36	28.8MB	1.84s	4.80s	12.7s	.380s	.506s	26.8s
GB48	38.4MB	4.53s	6.35s	16.8s	.505s	.672s	35.3s
GBPRI20	47.5MB	6.00s	7.78s	20.6s	.608s	.834s	43.3s
Throughput		6.5MB/s	6.2MB/s	2.3MB/s	80MB/s	57MB/s	1.1MB/s

Table 7.5: Memory Results for the Single Node

larger, reducing the transaction setup overhead. Similarly, the Osiris board's performance decreases because the data transfer sizes from the host processor to the board are decreased.

Figure 7.5 and Figure 7.6 show the sequence comparsion time versus the size of the database for the memory transfer method of communicating with the Osiris board. The slope of the line corresponds to the throughput of the system. The theoretical results shown are calculated using Equation 7.2 and Equation 7.4 for the local and remote systems.

Figure 7.7 and Figure 7.8 show the results of running the sequence comparing algorithm on the SLAAC-1V board. Included in the graphs are the theoretical results using the memory transfer model.

The Osiris board achieves better performance while streaming data than transferring data through memory. Using the FIFOs, the realized Osiris board implementation is capable

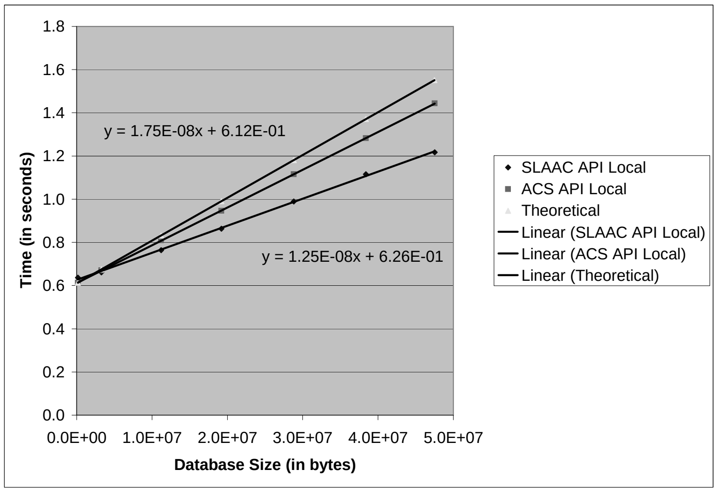


Figure 7.5: Single Local Osiris Board using Memory Transfers

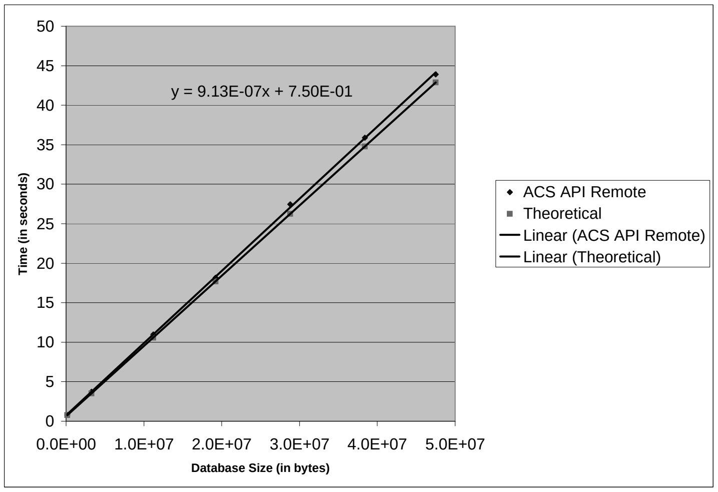


Figure 7.6: Single Remote Osiris Board using Memory Transfers

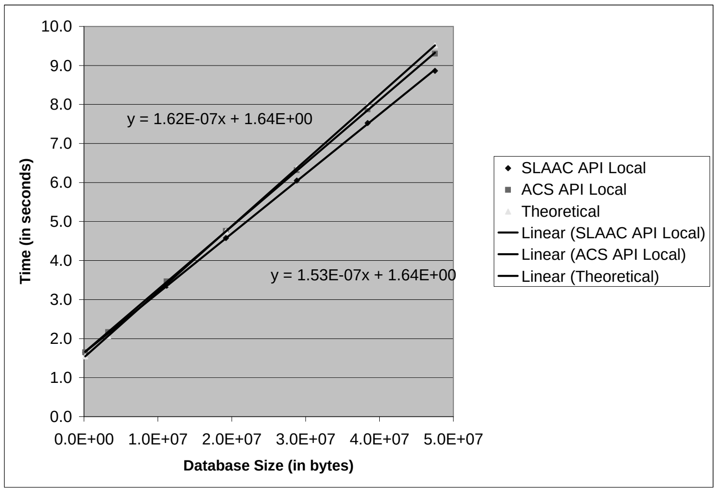


Figure 7.7: Single Local SLAAC-1V Board using Memory Transfers

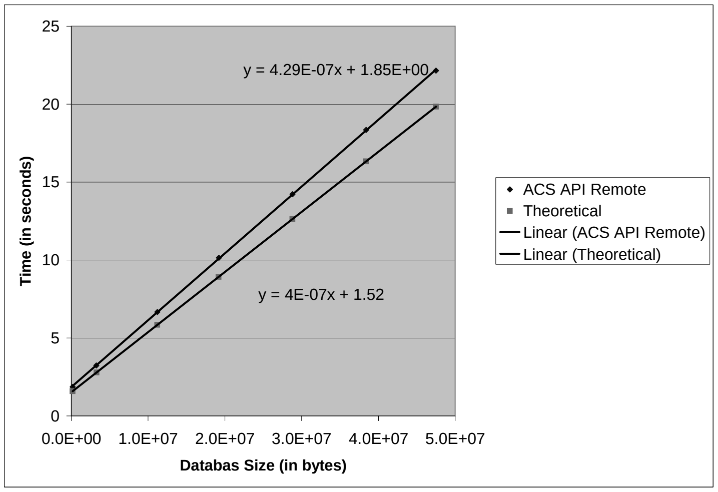


Figure 7.8: Single Remote SLAAC-1V Board using Memory Transfers

of 389 billion Smith-Waterman matrix cell updates per second compared to the theoretical 1.26 trillion cell updates per second. Conversely, the SLAAC-1V board handles memory transfers better than streamed data. Using the memory to transfer database sequences, the SLAAC-1V board is capable of 23 billion Smith-Waterman matrix cell updates per second compared to the theoretical 462 billion.

7.5 Streaming Approach

This section presents the results from the streaming approach to using multiple boards. When a gene sequence to be searched does not fit within a single board's configuration, the gene sequence is spread across multiple boards and the database is streamed through.

	Two Boards	Three Boards
GBUNA	.210s	.738s
GBPHG	4.21s	11.4s
GBGSS4	24.8s	67.1s
GBPRI20	162s	164s
System Throughput	.76MB/s	.59MB/s
CUPS	6.46B	7.97B

Table 7.6: Results for the Streaming Implementation

Table 7.6 presents the results for the streaming systems using the four genBank database sequences. In the streaming approach, the overall performance degrades to the throughput of the slowest node making this system a poor design for heterogeneous nodes. The two-board system uses the Osiris board in the host computer and streams data to the SLAAC-1V on the remote system. The speed of the system is clamped by the speed of the remote SLAAC-1V.

When a second SLAAC-1V board is added to the system, the performance degrades further. The degradation is due to the fact that the second SLAAC-1V system is even slower than the first. Both the 2-board and 3-board systems perform close to the speeds of the slowest node. An additional problem with this system is that the system can only scale to the size of the query sequence. Eighty percent of the sequences in the databases can be compared using a single configuration.

7.6 Distributed Approach

This section presents the results from the distributed approach to comparing DNA sequences. In each system, the host machine contains an Osiris board and the two other machines that contain SLAAC-1V boards have a network connection to the host. The database is streamed separately through each board. When a query sequence does not fit on one board, the output from the first configuration is stored in the host processors memory while the board is configured with the next part of the query sequence.

Table 7.7 shows the results for the distributed system using two and three boards. While the multiple node system has some performance gains, the network congestion limits the overall system performance. The Osiris board is on a separate PCI bus from the network card and allows the system to perform better than one that has a single PCI bus. However, each added node must vie for network bandwidth and since all traffic goes through the single network card, the host machine becomes a bottleneck. As seen in the table, adding the second board does not acheive the same performance gain as adding the first node.

	One Board	Two Boards	Three Boards
GBUNA	289B CUPS	267B CUPS	259B CUPS
GBPHG	379B CUPS	388B CUPS	393B CUPS
GBGSS4	380B CUPS	395B CUPS	397B CUPS
GBPRI20	380B CUPS	397B CUPS	398B CUPS

Table 7.7: Throughput for the Distributed Implementation

Chapter 8

Conclusions

8.1 Summary

This thesis presents the design and implementation of a gene matching algorithm developed on a distributed environment of adaptive computing boards using the ACS API. The development process was streamlined by implementing the system using the ACS API. Development tools such as the ACS textual and graphical interfaces allowed configurations to be debugged quickly and easily.

The sequence comparing hardware configuration was created and the benefits of using a run-time reconfigurable approach were shown. The configuration was optimized to increase the number of processing elements that fit in a single FPGA configuration.

Two designs for distributing FPGA boards in a system were presented and compared. In the streaming approach, the speed of the system is determined by the slowest node. This approach lends itself better to homogeneous systems. The distributed approach avoids the bottleneck of the slowest node in the system, allowing each board to run uninhibited. However, if the network shares the same bus as the FPGA board, then the overall system will slow down.

In conclusion, the ACS API is a simple programming environment that hides the complexity of a system. The increased usability does not add significant delays when using local nodes. The ACS API creates a uniformity among adaptive computing systems and allows tools to be developed that can be used on several different platforms. The gene matching application was able to be designed and created quickly using the ACS API.

8.2 Future Work

The gene matching systems developed and presented in this thesis compute the global edit distance. The next step is to develop an implementation that can compute and find interesting local edit distances. The local edit distance is a minima in the SW matrix. The local alignment implementation will pair counters with each processing element. Assuming 16-bit counters are used, two to three times fewer processing elements will fit in a local alignment FPGA configuration.

In addition to modifying the design presented in this thesis to compute local alignments, the design can be modified to compute edit distances for proteins as well. Proteins require five bits for each character. Therefore, the protein implementation will require at least two and a half times as much area as the DNA sequence matching implementation.

The gene matching configurations constructed in this thesis were developed statically prior to runtime. The final implementation requires runtime reconfiguration. At the time of this thesis, significant progress has been made to implement the runtime generation of configurations by manipulating a generic XDL file. JBits [33], once completed for Virtex-II parts, can be used for the runtime reconfiguration environment to further improve the design.

The ACS API performs well when using local nodes. However, a significant cost is incurred each time data must travel across the network. Many of the problems that occurred in increasing the performance across the network were a result of deficiencies in the drivers for the boards. More specifically, the ACS API requires that the FPGA drivers efficiently implement non-blocking functions for streaming data. Efforts have already been made to correct these problems; however, it is important that FPGA board designers recognize the needs of an automated environments such as the ACS API. Network traffic can be reduced, as well as increasing processing power, by expanding the ACS API to make use of the remote machines in the system. In this situation, the control processes resident on remote machines could be used to load, manipulate, and/or store information without contacting the host process. A single directive from the host process would trigger events on the remote systems, thus reducing the network traffic. The sequence comparing implementation described in this thesis could be run on several machines each with their own copy (or portion) of the database sequences.

Bibliography

[1] "The human genome project information page." http://www.genome.gov/, 2002.

[2] "National center for biotechnology information." http://www.ncbi.nlm.nih.gov/, 2002.

[3] "Swiss-prot protein knowledge." http://us.expasy.org/sprot/, 2002.

[4] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[5] C. Ulmer, *Configurable Computing: Practical Use of Field Programmable Gate Arrays.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, January 1999.

[6] Xilinx Inc., *The Programmable Logic Data Book.* San Jose, CA, 1999.

[7] "Slaac project home page." http://www.east.isi.edu/projects/SLAAC/, 2002.

[8] USC Information Sciences Institute, *SLAAC1-V User VHDL Guide*, 0.3.1 ed.

[9] P. B. B. Schott and L. Wang, *Osiris Board Architecture and VHDL Guide.* USC Information Sciences Institute, 1.1.1 ed., May 2002.

[10] X. Inc., *Virtex-II Platform FPGA Handbook.* San Jose, CA, December 2001.

[11] D. L. R. Lipton, "A systolic array for rapid string comparison," in *1985 Chapel Hill Conference on Very Large Scale Integeration* (H. Fuchs, ed.), pp. 363–376, 1985.

[12] M. Sievers, “The biologist’s guide to paracel’s similarity search algorithms.”

[13] “Fasta programs.” http://fasta.bioch.virginia.edu/, 2002.

[14] “Ncbi blast home page.” http://www.ncbi.nlm.nih.gov/BLAST/, 2002.

[15] “Decypher bioinformatics acceleration solution.” http://www.timelogic.com/decypher_intro.html, 2002.

[16] “Compugen.” http://www.cgen.com/, 2002.

[17] “Paracel, inc. world wide web site.” http://www.paracel.com/, 2002.

[18] “The genematcher2 system datasheet.” http://www.paracel.com/products/pdfs/gm2_datasheet.pdf, 2002.

[19] “Snort: The open source intrusion detection system.” http://www.snort.org/, 2002.

[20] R. Boyer and J. Moore, “A fast string searching algorithm,” *Communications of the ACM*, vol. 20, pp. 762–772, October 1977.

[21] N. K. Ratha, K. Karu, S. Chen, and A. K. Jain, “A real-time matching system for large fingerprint databases,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 18, no. 8, pp. 799–813, 1996.

[22] N. K. Ratha, A. K. Jain, and D. T. Rover, “Fingerprint matching on splash 2,” Tech. Rep. MSU-CPS-99-20, Department of Computer Science, Michigan State University, East Lansing, Michigan, April 1999.

[23] L. Rabiner, “A tutorial on hidden markov models and selected applications in speech recognition,” in *Proceedings of the IEEE*, vol. 77, February 1989.

[24] R. Hughey, “Massively parallel biosequence analysis,” Tech. Rep. UCSC-CRL-93-14, 1993.

[25] O. Gotoh, "An improved algorithm for matching biological sequences," *Jornal of Molecular Biology*, vol. 162, pp. 705–708, 1982.

[26] M. Jones, L. Schraf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, and P. Athanas, "Implementing an API for distributed adaptive computing systems," in *IEEE Symposium on FPGAs for Custom Computing Machines* (K. L. Pocek and J. Arnold, eds.), (Los Alamitos, CA), pp. 222–230, IEEE Computer Society Press, 1999.

[27] K. Yao, "Implementing an application programming interface for distributed adaptive computing systems," Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, May 2000.

[28] "Extensible markup language (xml)." http://www.w3.org/XML/, 2002.

[29] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra, *MPI: The Complete Reference*, 1997.

[30] "Tcldeveloper site." http://www.tcl.tk/, 2002.

[31] "Trolltech: Creators of qt, the cross-platform c++ gui toolkit." http://www.tcl.tk/, 2002.

[32] S. Guccione and E. Keller, "Gene matching using jbits." 2002.

[33] S. Guccione and D. Levi, "XBI: A Java-based interface to FPGA hardware," in *Proc. SPIE Photonics East, J. Schewel (Ed.), SPIE - The International Society for Optical Engineering, Bellingham, WA*, November 1998.

Appendix A

Hokiegene Code

This appendix contains code from the gene sequence comparison implementation described in this thesis. Listing A.1 shows the hardware description of a processing element for the Adenine nucleotide. Listing A.2 shows the system configuration file for the streaming system. Listing A.3 shows the host program code for the streaming system. Listing A.4 shows the system configuration file for the distributed system. The final listing, Listing A.5, contains the host program code for the distributed system.

Listing A.1: Processing Element Hardware Description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
library slaac;
use slaac.seq_pkg.all;
entity seq_proc_a0 is
    port (
        clk     : in std_logic;                         -- global clock
        RSTin   : in std_logic;                         -- synchronous reset signal (input)
        TChin   : in std_logic_vector (1 downto 0);     -- database DNA (input)
        EDITin  : in std_logic;                         -- edit distance (input)
        RSTout  : out std_logic;                        -- synchronous reset signal (output)
        TChout  : out std_logic_vector (1 downto 0);    -- database DNA (output)
        EDITout : out std_logic                         -- edit distance (output)
        );
end seq_proc_a0;
architecture initial of seq_proc_a0 is
        signal A0, A0_n, A1, A1_n, B, C, dout, dout_int, comp1_1, comp1_0,
                comp0_1, comp0_0, data0_1, z0   :  std_logic;
        component comp1 is                      -- Compares 2 DNA characters
                port (
                t1in, t0in, s1in, s0in  : in std_logic;
                comp    : out std_logic
                );
        end component;
        component comp2 is                      -- Compares values in cells a, b, and c
                port (
                a1, a0, b, c    : in std_logic;
                comp    :out std_logic
                );
        end component;
begin
        --Slice 3
        -- pass the synchronous reset through
     lrststore  :  dff_s  port map (clk => clk, set => RSTin, Din => RSTin,
                 Dout => RSTout);
        -- compute cell 'a' + 2 value
     lA1_n : inv port map (a => A1, b => A1_n);
        -- send either 'a + 2' or value from comp1_1
     ldout_int  :  xor_2 port map (a => A1_n, b => comp1_1, c => dout_int);
        -- Flip-Flop for edit distance
     ldout    :  dff_r  port map (clk => clk, rst => RSTin, Din => dout_int,
                 Dout => dout);
        -- Send edit distance to next processing element
     EDITout <= dout;

     --Slice 2
        -- signal mismatch or send value form comp0_1
     lcomp11 : MUXCY port map (o => comp1_1, DI => '1', CI => comp0_1,
                 S => comp1_0);
        -- compare database character (TChin) to query character (sXin)
     lcomp10 : comp1 port map (t1in => TChin(1), t0in => TChin(0), s1in => '0',
                 s0in => '0', comp => comp1_0);
        -- pass the database character through
     lt1out   :  dff_r  port map (clk => clk, rst => RSTin, Din => TChin(1),
                 Dout => TChout(1));
        -- signal to use values for cell 'a', 'b', or 'c'
     lcomp01 : MUXCY port map (o => comp0_1, DI => '1', CI => '0',
                 S => comp0_0);
        -- compare 'a+1', b, and c to determine minimum value to send
      lcomp00 : comp2 port map (a1 => A1, a0 => A0, b => dout, c => EDITin,
            comp => comp0_0);
        -- pass the database character through
      lt0out  :  dff_r  port map (clk => clk, rst => RSTin, Din => TChin(0),
            Dout => TChout(0));

      --Slice 1
        -- store cell 'b' for use as cell 'a' in next time step
      lA1 :  dff_r  port map (clk => clk, rst => RSTin, Din => EDITin,
            Dout => A1);
        -- construct lower bit of cell 'a' (toggles due to scoring format)
      lA0_n : inv port map (a => A0, b => A0_n);
        -- store cell 'b' for use as cell 'a' in next time step
      lA0 :  dff_r  port map (clk => clk, rst => RSTin, Din => A0_n, Dout => A0);
end initial;
```

Listing A.2: Streaming System XML Configuration File

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE config SYSTEM "/project/acs_api/src/config/acs_conf.dtd">
<config>
<header>
   <commtype>MPI</commtype>
   <version>0.03</version>
   <author>William J. Worek</author>
   <desc>Streaming System Configuration</desc>
</header>
```

```
<boards>
  <osiris board_id="node0" location="local" ctrl_proc_dir ="/project/acs_api/bin">
    <PE pe_num="0" prog_file="xp_es.bit"/>
  </osiris>
  <slaac1v board_id="node1" location="xanadu" ctrl_proc_dir="/project/acs_api/bin">
    <PE pe_num="0" prog_file="x0.bit"/>
    <PE pe_num="1" prog_file="x1.bit"/>
    <PE pe_num="2" prog_file="x2.bit"/>
  </slaac1v>
  <slaac1v board_id="node2" location="cowbell" ctrl_proc_dir="/project/acs_api/bin">
    <PE pe_num="0" prog_file="x0.bit"/>
    <PE pe_num="1" prog_file="x1.bit"/>
    <PE pe_num="2" prog_file="x2.bit"/>
  </slaac1v>
</boards>
<channels>
  <channel>
    <src><host port="HOST_OUT"/></src>
    <dest><node board_id="node1" port="SLAAC_FIFO_A1"/></dest>
  </channel>
  <channel>
    <src><node board_id="node1" port="SLAAC_FIFO_B1"/></src>
    <dest><node board_id="node2" port="SLAAC_FIFO_A1"/></dest>
  </channel>
  <channel>
    <src><node board_id="node2" port="SLAAC_FIFO_B1"/></src>
```

```
        <dest><node board_id="node0" port="OSIRIS_FIFO_A0"/></dest>
    </channel>
    <channel>
        <src><node board_id="node0" port="OSIRIS_FIFO_B0"/></src>
        <dest><host port="HOST_IN"/></dest>
    </channel>
</channels>
</config>
```

Listing A.3: Streaming System Host Program

```
/* ************************************************************************
 * This program for a streaming structure of FPGA boards performing the
 *     Smith-Waterman global comparison
 *     Usage: fifo_test  xml_config_file  gene_file
 * Virginia Tech Configurable Computing Laboratory
 * Copyright 2002   William J. Worek
 ************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fstream.h>
#include <iostream.h>
#include "acs.h"
#include "acs_system.h"
#include "acs_xmlconfig.h"
/************************************************************************/
/*      Constants                                                        */
```

```
/*********************************************************************/
#define CLOCK_FREQUENCY 50
#define BUF_SIZE 0x7fffff
/*********************************************************************/
/*      Data Structures                                              */
/*********************************************************************/
struct FIFOWORD {
  unsigned int top;
  unsigned int bottom;
};
/*********************************************************************/
/* Dequeue function                                                  */
/* Loops until all the information is dequeued                       */
/*********************************************************************/
inline void my_dequeue(unsigned char * in_buffer, int size, int port,
    ACS_SYSTEM * system, ACS_STATUS status) {
  int dequeue = 0;
  int dval;
  while(dequeue < size)
  {
    dval = ACS_Dequeue(&(in_buffer[dequeue]), size-dequeue, port, system, &status);
    dequeue += dval;
  }
}
/*********************************************************************/
/* Main Routine                                                      */
```

```
/*******************************************************************/
void main(int argc, char ** argv)
{
   FIFOWORD fifo_input, fifo_output;   // FIFOWORD formatted data
   ACS_STATUS status;                  // Status of ACS function
   ACS_SYSTEM * system;                // ACS system object
   ACS_CLOCK clock;                    // ACS clock object
   ACS_XMLConfig config;               // object for configuring from XML
   ACS_BOARD_INFO board_info[3];       // Board information
   int  num_sites;                     // Number of nodes in the system
   int  node_id, node_num;             // ID and index of node
   int pe_mask, num_pes;               // FPGA board processing element data
   int  rc;                            // Return code for ACS functions
   int  i, j;                          // Loop indices
   int num_sent;                       // Size of datbase
   int thread_status;                  // Thread status information
   char filename[40];                  // database file name
   unsigned char *out_buffer;          // database buffer
   unsigned char *in_buffer;           // processed data buffer
   out_buffer = new unsigned char [BUF_SIZE*8];
   in_buffer = new unsigned char [BUF_SIZE*8];
   ifstream  infile ;                  // database file
   int enport = 0;                     // Enqueue port
   int deport = 1;                     // Dequeue port
   int send, sendsize;                 // Amount sent; block size to send
     // Parse command line parameters
```

```
  // Get database file name
 if (argc == 3)
    strcpy(filename, argv[2]);
 else
 {
    printf ("USAGE: %s xml_file gene_file\n", argv[0]);
    return 1;
 }
 strcat(filename ,".gen");
//-------------------------------------
//-- Construct and start ACS system
//-------------------------------------
  // Set XML configuration file name
 printf ("Configuring the system using: %s\n", argv[1]);
 config.setFilename(argv[1]);
  // Get system object from XML configuration system
 if (config.getSystem(&system) != ACS_SUCCESS)
 {
    printf ("Failed to create the system object, terminating\n");
    return 2;
 }
  // Get number of nodes in the system
 if (config.getNodeCount(&num_sites) != ACS_SUCCESS)
    printf ("Failed to get the number of nodes\n");
  // Start the clocks on the nodes in reverse order
 for (node_num = num_sites - 1; node_num >= 0; node_num--)
```

```
{
    // Get number and information for this particular node
  ACS_Get_Board_Info(system, node_num, &board_info[node_num]);
  if ((rc = config.getNodeNumber(node_num, &node_id)) != ACS_SUCCESS)
    printf (" Failed to get number for node: %d\n", node_num);
  pe_mask = 0;
  for (i = 0; i < board_info[node_num].num_pes; i++)
    pe_mask = pe_mask | board_info[node_num].pe_mask[i];
    // Set the frequency of the clock on this node
  clock.frequency = CLOCK_FREQUENCY;
  clock.countdown = 0;
  if ((rc = ACS_Clock_Set(&clock, node_id, system, &status)) != ACS_SUCCESS)
    printf (" Failure in ACS_Clock_Set, rc = %d\n", rc);
    // Assert reset signal
  if ((rc = ACS_Reset(node_id, system, pe_mask, 1, &status)) != ACS_SUCCESS)
    printf (" Failure in ACS_Reset, rc = %d\n", rc);
    // Start the board
  if ((rc = ACS_Run(node_id, system, &status)) != ACS_SUCCESS)
    printf (" Failure in ACS_Run, rc = %d\n", rc);
    // Deassert reset signal
  if ((rc = ACS_Reset(node_id, system, pe_mask, 0, &status)) != ACS_SUCCESS)
    printf (" Failure in ACS_Reset, rc = %d\n", rc);
}
//------------------------------------
//-- Read and process database
//------------------------------------
```

```
infile.open(filename,ios::in);
while (!infile.eof()) {
    // Read in database from file
    num_sent = 0;
    infile >> fifo_input.top >> fifo_input.bottom;
    while ((!infile.eof()) && (num_sent < BUF_SIZE))
    {
        ((unsigned int *) outbuffer)[2*num_sent] = fifo_input.top;
        ((unsigned int *) outbuffer)[2*num_sent+1] = fifo_input.bottom;
        num_sent++;
        infile >> fifo_input.top >> fifo_input.bottom;
    }
    // Stream data through system in blocks
    send = 0;
    sendsize = 0;
    count = 0;
    while (send < num_sent)
    {
        if((num_sent-send) > 2500)
            sendsize = 2500;
        else
            sendsize = num_sent-send;
        ACS_Enqueue(&out_buffer[send*8], 8*sendsize, enport, system, &status);
        my_dequeue(in_buffer, 8*sendsize, deport, system, status);
        send += sendsize;
    }
    }
     infile .close ();
    //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    //−− Free memory and destroy ACS system
    //−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    delete(out_buffer);
    delete(in_buffer );
      // Destroy the acs_system, release the resource.
    if ((rc = ACS_System_Destroy(system, &status)) != ACS_SUCCESS)
        printf (" Call to ACS_System_Destroy unsuccessful, error: %d\n", rc);
      // finalize the acs world, destroy throughly.
    if ((rc = ACS_Finalize()) != ACS_SUCCESS)
        printf (" Call to ACS_Finalize unsuccessful, error : %d\n", rc);
    printf (" VPI & SU, ECPE, CCM Lab. Spring 2001\n");
}
```

Listing A.4: Distributed System XML Configuration File

```
<?xml version='1.0' encoding='UTF−8'?>
<!DOCTYPE config SYSTEM "/project/acs_api/src/config/acs_conf.dtd">
<config>
<header>
    <commtype>MPI</commtype>
    <version>0.03</version>
    <author>William J. Worek</author>
    <desc>Distributed System Configuration</desc>
</header>
<boards>
```

```
    <osiris board_id="node0" location="local" ctrl_proc_dir="/project/acs_api/bin">
        <PE pe_num="0" prog_file="xp_es.bit"/>
    </osiris>
    <slaac1v board_id="node1" location="xanadu" ctrl_proc_dir="/project/acs_api/bin">
        <PE pe_num="0" prog_file="x0.bit"/>
        <PE pe_num="1" prog_file="x1.bit"/>
        <PE pe_num="2" prog_file="x2.bit"/>
    </slaac1v>
    <slaac1v board_id="node2" location="cowbell" ctrl_proc_dir="/project/acs_api/bin">
        <PE pe_num="0" prog_file="x0.bit"/>
        <PE pe_num="1" prog_file="x1.bit"/>
        <PE pe_num="2" prog_file="x2.bit"/>
    </slaac1v>
</boards>
<channels>
    <channel>
        <src><host port="0"/></src>
        <dest><node board_id="node0" port="OSIRIS_FIFO_A0"/></dest>
    </channel>
    <channel>
        <src><node board_id="node0" port="OSIRIS_FIFO_B0"/></src>
        <dest><host port="1"/></dest>
    </channel>
    <channel>
        <src><host port="2"/></src>
        <dest><node board_id="node1" port="SLAAC_FIFO_A1"/></dest>
```

```
  </channel>
  <channel>
      <src><node board_id="node1" port="SLAAC_FIFO_B1"/></src>
      <dest><host port="3"/></dest>
  </channel>
  <channel>
      <src><host port="4"/></src>
      <dest><node board_id="node2" port="SLAAC_FIFO_A1"/></dest>
  </channel>
  <channel>
      <src><node board_id="node2" port="SLAAC_FIFO_B1"/></src>
      <dest><host port="5"/></dest>
  </channel>
</channels>
</config>
```

Listing A.5: Distributed System Host Program

```
/* *********************************************************************
 * This program for a distributed structure of FPGA boards performing the
 *     Smith-Waterman global comparison
 *     Usage: fifo_test  xml_config_file  gene_file
 * Virginia Tech Configurable Computing Laboratory
 * Copyright 2002    William J. Worek
 *********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "acs.h"
#include "acs_system.h"
#include "acs_xmlconfig.h"
#include <fstream.h>
#include <iostream.h>
#include <pthread.h>
/*****************************************************************/
/*      Constants                                                */
/*****************************************************************/
#define CLOCK_FREQUENCY 50 //Clock frequency for the FPGA boards
#define BUF_SIZE 0x7fffff //Buffer size for the database file
#define MEM_SIZE 250000 //Size of the Addressable Memory (SLAAC-1V)
/*****************************************************************/
/*      Data Structures                                          */
/*****************************************************************/
struct FIFOWORD {
   unsigned int top;
   unsigned int bottom;
};
struct thread_data{
   int num_sent;
   int node_id;
   ACS_SYSTEM * system;
   unsigned char *out_buffer;
};
/*****************************************************************/
```

```
/* Dequeue function                                                    */
/* Loops until all the information is dequeued                          */
/**********************************************************************/
inline void my_dequeue(unsigned char * in_buffer, int size, int port,
        ACS_SYSTEM * system, ACS_STATUS status) {
    int dequeue = 0;
    int dval;
    while(dequeue < size)
    {
        dval = ACS_Dequeue(&(in_buffer[dequeue]), size-dequeue, port, system, &status);
        dequeue += dval;
    }
}
/**********************************************************************/
/* Thread for the Osiris Board.                                         */
/* Passes the database through the board using FIFOs.                   */
/**********************************************************************/
void osiris (void * threadarg)
{
    ACS_SYSTEM * system;       // ACS system Object
    ACS_STATUS status;         // Status of function call
    struct thread_data * my_data;// Place holder for thread parameters
    unsigned char * out_buffer; // database buffer
    int num_sent;              // number of words in the database buffer
    int i;                     // loop index variable
    int enport = 0;            // enqueue port
```

```
   int deport = 1;                // dequeue port
   int node_id;                   // ID value of osiris node
   unsigned char *in_buffer;      // buffer for the output of the board
   in_buffer = new unsigned char [BUF_SIZE*8];
     // Reconstruct parameters from the threadarg
   my_data = (struct thread_data *) threadarg;
   system = my_data->system;
   out_buffer = my_data->out_buffer;
   num_sent = my_data->num_sent;
   node_id = my_data->node_id;
     //Enqueue database and Dequeue Results
   ACS_EnqueueNode(out_buffer, 8*num_sent, node_id, enport, system, &status);
   my_dequeueNode(in_buffer, 8*num_sent, node_id, deport, system, status);
   delete(in_buffer);
   pthread_exit(0);
}
/*********************************************************************/
/* Thread for the SLAAC-1V Board.                                  */
/* Passes the database through the board using Memories.           */
/*********************************************************************/
void slaac1v (void * threadarg)
{
   ACS_SYSTEM * system;       // ACS system Object
   ACS_STATUS status;         // Status of function call
   ACS_ADDRESS address;
   struct thread_data * my_data;// Place holder for thread parameters
```

```
unsigned char * out_buffer; // database buffer
int num_sent;               // number of words in the database buffer
int i;                      // loop index variable
int send, sendsize;         // amount of data sent; size of block to send
int enport = 0;             // enqueue port
int deport = 1;             // dequeue port
int node_id;                // ID value of osiris node
unsigned char *in_buffer;   // buffer for the output of the board
in_buffer = new unsigned char [BUF_SIZE*8];
  // Reconstruct parameters from the threadarg
my_data = (struct thread_data *) threadarg;
system = my_data->system;
out_buffer = my_data->out_buffer;
num_sent = my_data->num_sent;
node_id = my_data->node_id;
enport = (node_id-1)*2;
deport = enport+1;
sendsize = MEM_SIZE;
send = 0;
  // Loop through the database buffer
while (send < num_sent)
{
   address.pe = 2;
   address.mem = 0;
   address.offset = 0;
     // Load Memory with portion of database
```

```
        ACS_Write((void *) &out_buffer[send], sendsize, 4, node_id, &address, system,
                &status);
          // Signal FPGA to pass memory through the system
        ACS_Enqueue(in_buffer, 8, enport, system, &status);
          // Wait for FPGA to complete processing
        my_dequeue((unsigned char *)in_buffer, 8, deport, system, status);
        address.mem = 1;
          // Read the computed values from memory
        ACS_Read((void *) in_buffer, sendsize, 4, node_id, &address, system, &status);
        send += sendsize;
    }
    delete(in_buffer);
    pthread_exit(0);
}
/*********************************************************************/
/* Main Routine                                                     */
/*********************************************************************/
void main(int argc, char ** argv)
{
    FIFOWORD fifo_input, fifo_output;   // FIFOWORD formatted data
    thread_data thread_data_array[3];   // Data for sending to threads
    ACS_STATUS status;                  // Status of ACS function
    ACS_SYSTEM * system;                // ACS system object
    ACS_CLOCK clock;                    // ACS clock object
    ACS_XMLConfig config;               // object for configuring from XML
    ACS_BOARD_INFO board_info[3];       // Board information
```

```
int  num_sites;                     // Number of nodes in the system
int  node_id, node_num;             // ID and index of node
int pe_mask, num_pes;               // FPGA board processing element data
int  rc;                            // Return code for ACS functions
int  i , j ;                        // Loop indices
int num_sent;                       // Size of datbase
int thread_status;                  // Thread status information
char filename[40];                  // database file name
unsigned char *out_buffer;          // database buffer
out_buffer = new unsigned char [BUF_SIZE*8];
ifstream  infile ;                  // database file
pthread_t  isis_thread , xanadu_thread, cowbell_thread; // disritbuted threads
  // Parse command line parameters
  // Get database file name
if (argc == 3)
   strcpy(filename, argv [2]);
else
{
    printf (" USAGE: %s xml_file gene_file\n", argv[0]);
   return 1;
}
strcat(filename ,". gen");
//----------------------------------------
//-- Construct and start ACS system
//----------------------------------------
  // Set XML configuration file name
```

```
printf ("Configuring the system using: %s\n", argv[1]);
config.setFilename(argv[1]);
  // Get system object from XML configuration system
if (config.getSystem(&system) != ACS_SUCCESS)
{
    printf ("Failed to create the system object, terminating\n");
    return 2;
}
  // Get number of nodes in the system
if (config.getNodeCount(&num_sites) != ACS_SUCCESS)
    printf ("Failed to get the number of nodes\n");
  // Start the clocks on the nodes in reverse order
for (node_num = num_sites - 1; node_num >= 0; node_num--)
{
      // Get number and information for this particular node
    ACS_Get_Board_Info(system, node_num, board_info[node_num]);
    if ((rc = config.getNodeNumber(node_num, &node_id)) != ACS_SUCCESS)
        printf ("Failed to get number for node: %d\n", node_num);
    pe_mask = 0;
    for (i = 0; i < board_info[node_num].num_pes; i++)
        pe_mask = pe_mask | board_info[node_num].pe_mask[i];
      // Set the frequency of the clock on this node
    clock.frequency = CLOCK_FREQUENCY;
    clock.countdown = 0;
    if ((rc = ACS_Clock_Set(&clock, node_id, system, &status)) != ACS_SUCCESS)
        printf ("Failure in ACS_Clock_Set, rc = %d\n", rc);
        // Assert reset signal
      if ((rc = ACS_Reset(node_id, system, pe_mask, 1, &status)) != ACS_SUCCESS)
          printf ("Failure in ACS_Reset, rc = %d\n", rc);
        // Start the board
      if ((rc = ACS_Run(node_id, system, &status)) != ACS_SUCCESS)
          printf ("Failure in ACS_Run, rc = %d\n", rc);
        // Deassert reset signal
      if ((rc = ACS_Reset(node_id, system, pe_mask, 0, &status)) != ACS_SUCCESS)
          printf ("Failure in ACS_Reset, rc = %d\n", rc);
}
//-----------------------------------------
//-- Read and process database
//-----------------------------------------
infile.open(filename,ios::in);
while (!infile.eof()) {
        // Read in database from file
    num_sent = 0;
    infile >> fifo_input.top >> fifo_input.bottom;
    while ((!infile.eof()) && (num_sent < BUF_SIZE))
    {
        ((unsigned int *) outbuffer)[2*num_sent] = fifo_input.top;
        ((unsigned int *) outbuffer)[2*num_sent+1] = fifo_input.bottom;
        num_sent++;
        infile >> fifo_input.top >> fifo_input.bottom;
    }
        // Pack parameters for sending to thread
    for (i = 0; i < 3; i++)
    {
        thread_data_array[i].out_buffer = out_buffer;  //pointer to database
        thread_data_array[i].system = system;          //pointer to system
        thread_data_array[i].num_sent = num_sent;      //size of database
        thread_data_array[i].node_id = i;              //node ID
    }
      // Process database on each node
    pthread_create(&cowbell_thread, NULL, (void * (*)(void *)) slaac1v,
            (void *) &thread_data_array[2]);
    pthread_create(&xanadu_thread, NULL, (void * (*)(void *)) slaac1v,
            (void *) &thread_data_array[1]);
    osiris (&thread_data_array[0]);
      // Wait for each node to complete
    pthread_join(cowbell_thread,(void **) &thread_status );
    pthread_join(xanadu_thread,(void **) &thread_status );
}
infile.close();
//-----------------------------------------
//-- Free memory and destroy ACS system
//-----------------------------------------
delete(out_buffer);
  // Destroy the acs_system, release the resource.
if ((rc = ACS_System_Destroy(system, &status)) != ACS_SUCCESS)
    printf ("Call to ACS_System_Destroy unsuccessful, error: %d\n", rc);
  // finalize the acs world, destroy throughly.
```

```
    if ((rc = ACS_Finalize()) != ACS_SUCCESS)
        printf ("Call to ACS_Finalize unsuccessful, error: %d\n", rc);
    printf ("VPI & SU, ECPE, CCM Lab. Spring 2002\n");
}
```

Vita

William J. Worek was born and raised in Clifton, a small town in Virginia. He matriculated to Virginia Tech in 1995 after completing his high school education at Thomas Jefferson High School for Science and Technology. In 1999, he received a summa cum laude Bachelor's degree in honors in Computer Engineering. During his undergraduate career, William spent a semester working at Northern Telecom in the XPM Diagnostic group. William decided to continue at Virginia Tech for his Masters Degree in Computer Engineering. As a graduate student, William worked in the Configurable Computing Machines laboratory developing an API for distributed adaptive computing systems. He finds time to enjoy the outdoors by skiing, playing tennis, and hiking. William spends a lot of time giving back to the community as a Life Member of the Virginia Tech Rescue Squad.

Reconfigurable Architectures for Systems Level Applications of Adaptive Computing

Brian Schott, Steve Crago, Chen Chen,
Joe Czarnaski, Matt French, Ivan Hom[*], Tam Tho, and Terri Valenti

University of Southern California Information Sciences Institute
4350 N. Fairfax Drive, Suite 770, Arlington, VA 22203

ABSTRACT

The Systems Level Applications of Adaptive Computing (SLAAC) project is defining an open, distributed, scalable, adaptive computing systems architecture based on a high-speed network cluster of heterogeneous, FPGA-accelerated nodes. Two implementations of this architecture are being created. The Research Reference Platform (RRP) is a Myrinet™ cluster of PCs with SLAAC-1 PCI-based FPGA accelerators. The Deployable Reference Platform (DRP) is a Myrinet cluster of PowerPC nodes with SLAAC-2 VME-based FPGA accelerators. A commercial 6U-VME quad-PowerPC board, the CSPI M2641™, has been adapted to act as a carrier for SLAAC-2. A key strategy proposed for successful ACS technology insertions is source-code compatibility between the RRP and DRP platforms. This paper focuses on the development of the SLAAC-1 and SLAAC-2 accelerators and how the network-centric SLAAC system-level architecture has shaped their designs. A preliminary mapping of a Synthetic Aperture Radar / Automatic Target Recognition (SAR/ATR) algorithm to SLAAC-2 is also discussed.

KEY WORDS

FPGA, SLAAC, adaptive computing, reconfigurable computing, high performance computing, SAR/ATR

INTRODUCTION

The mission of the Systems Level Applications of Adaptive Computing (SLAAC) project is to: 1) define an open, distributed, scalable, adaptive computing systems architecture; 2) design, develop, and evolve scalable reference platform implementations of this architecture; and 3) validate the approach by deploying technology in multiple defense application domains. In the context of this research, adaptive computing systems (ACS) refer to systems that reconfigure their logic and/or data paths in response to dynamic application requirements. Creation and validation of a scalable, distributed, ACS architectures requires a closely coordinated hardware and software development effort in the areas of next-generation FPGA accelerators, module generators and other tools, runtime control libraries and APIs, and algorithm mapping. Although the SLAAC team is presently engaged in all of these activities, this paper focuses on the SLAAC-1 and

[*] University of Southern California Information Sciences Institute,
4676 Admiralty Way, Suite 1000, Marina del Rey, CA 90292

SLAAC-2 FPGA accelerator hardware design efforts currently underway for the first generation reference platforms.

The system-level focus of the SLAAC project came about because of the realization that scalability and portability are the two primary obstructions preventing innovative ACS research from being directly useful in deployed systems. Scalability is an issue in that many real-world applications are larger than the modern PCI-based ACS accelerator. Transitioning from a small proof of concept demonstration to large real-world application is often overlooked in ACS research. The portability issue has both a hardware and software aspect. Physical form-factor and operating system issues can limit the utility of algorithm research on PCI-based FPGA accelerators. For example, a deployed system may require VME-based hardware and a real-time OS such as VxWorks™. Even among strictly PCI-based accelerators there is little commonality in the hardware architectures and software environments; rehosting to a field-friendly platform may be as difficult as the original research.

The SLAAC approach to scalability leverages modern cluster-computing techniques. The cluster computing community uses workstations and COTS high-speed network "back-plane" to build high-performance parallel systems [1]. Therefore, a logical way to build scalable parallel ACS systems is to cluster FPGA-accelerated workstations. This workstation-based architecture is called a Research Reference Platform (RRP). The SLAAC team is developing a scalable API and runtime software to support application control of these network-distributed multiple-host multiple-board ACS systems. The RRP has the advantage of being an inexpensive readily available platform for ACS development that tracks advances in workstations, adaptive computing, and cluster computing. The Tower of Power (ToP) at Virginia Tech is a good example of an RRP. The ToP has sixteen Pentium II™ PCs each equipped with a WildForce™ board tightly coupled to a Myricom™ LAN/SAN card; the PCs are connected through a sixteen port Myrinet™ switch. A total of 80 XC4062XL FPGAs and memory banks are distributed throughout the platform, and are available as computing resources [2].

Portability to deployable systems is partially addressed by implementing this same ACS-accelerated cluster architecture in a field-friendly platform. The scalable API and runtime control software being developed for the RRP are also under development in a VxWorks environment for Single Board Computers (SBCs). This field-friendly version of the RRP is called a Deployable Reference Platform (DRP). Scalable source-code compatibility can be achieved with respect to the host processor application. However, VHDL-level or bit-file level compatibility of the application code on the FPGA-based accelerator requires an identical FPGA architecture on both the RRP and DRP platforms. For this purpose, the SLAAC team is developing two FPGA accelerators. The SLAAC-1 board is a standard full-sized 64-bit PCI board intended for RRP workstations. SLAAC-2 is a 6U VME mezzanine board designed to be plugged into a modified CSPI 2641 Quad PowerPC baseboard. From the application perspective, the SLAAC-1 and SLAAC-2 boards represent the same ACS architecture. Section 2 of this paper discusses this common hardware architecture in detail. Details relating to the specific SLAAC-1 and SLAAC-2 implementations are described in Section 3. Hardware support software such as device drivers and a control library is briefly covered in Section 4. A preliminary application mapping to the SLAAC architecture is covered in Section 5, and future work is discussed in section 6.

SLAAC ARCHITECTURE

The SLAAC architecture is an attached processor system comprised of FPGAs and fast local memories. The basic concept isn't significantly changed from predecessor reconfigurable computer architectures such as Splash 2 [3] and Wildforce™ [4]. As shown in Figure 1 the SLAAC-1 architecture is partitioned into a single interface FPGA (labeled 'IF') and three user-programmable FPGAs (labeled 'X0', 'X1', and 'X2'). The IF chip is configured at power-up to act as a stable bridge to the host system bus. It provides configuration, clock, and control logic for the user FPGAs. The attached host is responsible for actually programming the user FPGAs and controlling the system. SLAAC-1 is designed to act either synchronously with the host, or asynchronously with DMA channels transporting data to and from host memory. A clock generator and FIFOs implemented within IF allow the user FPGAs to operate from a single data-synchronous clock in either mode of operation.

Data Paths

One of our goals with the SLAAC-1 architecture was to design an FPGA accelerator assuming a 64-bit data word. Since fast 64-bit system busses have become more commonplace in commodity PCs, we felt that a 64-bit data word was necessary to keep up with modern I/O rates. A 64-bit word is also a more natural atomic data element for these wider processors and even/odd word alignment issues of a 32-bit FPGA system would cause additional complexity in user FPGA designs. Consequently, the two bi-directional 72-bit "FIFO" connections between IF and X0 permit the user FPGAs to produce and consume a 64-bit data word in a single clock cycle. The three user-programmable FPGAs are organized in a ring structure. X0 acts as the control element for managing user data flow, thus enabling X1 and X2 to focus on computation. The ring path (X0→X1→X2→X0) is also 72-bits wide so that an 8-bit tag can be associated with each 64-bit data word. The individual pin directions on the ring connections are user-controlled; this architecture could just as easily support one 36-bit clockwise ring, and one 36-bit counterclockwise ring. The "crossbar" connecting X0, X1, and X2 together is a common 72-bit bus. The user also controls the direction of individual pins of this crossbar. Six additional handshake lines not shown (two each from X0 to X1, from X1 to X2, and from X0 to X2) permit crossbar arbitration without requiring unique configurations in X1 and X2.

Processing Elements

The SLAAC processing elements X1 and X2 each consist of one Xilinx XC40150XV-09 FPGA and four 256Kx18bit synchronous SRAMs. The Xilinx 40150 contains a 72x72 array of CLBs for 100K to 300K equivalent logic gates supporting clock speeds up to 100MHz. The SRAMs feature zero-bus turnaround permitting a read or write every cycle; no idle cycles are required for write after read with the only tradeoff being that writes are pipelined [5]. Each PE has two 72-bit connections to left and right neighbors for systolic data and a 72-bit connection to the shared crossbar. Other connections not shown include four LED lines, two handshake lines connected to X0, and miscellaneous reset, clock, configuration, and readback pins.

The location of the memories and the major connections are designed to permit the PE to be divided into four "Splash-2-like" single-memory systolic processors to improve pipelining and floor planning. The memories are arranged along the top of the PE, the crossbar connection is centered on the bottom, and the left and right ring connections are on the left and right sides respectively. Both X1 and X2 processing elements are identical so that the same SIMD or systolic configuration can be easily replicated without redundant synthesis.

Control Element

The SLAAC control element, X0, consists of one Xilinx XC4085XLA-09 and two 256Kx18bit synchronous SRAMs. The Xilinx 4085 contains a 56x56 array of CLBs for a 55K to 180K equivalent gates at clock rates up to 100MHz. X0 has two 72-bit ring connections, a 72-bit shared crossbar connection, and two 72-bit FIFO connections to the interface FPGA. Unlike the Splash 2 and WildForce™, X0 in the SLAAC architecture is designed to sit at both ends of the systolic array. X0 acts as the data stream manager for the architecture. Its primary mission is to read/write data from the FIFO module blocks implemented in the IF chip and pass this data on to the processing elements. The location of the memories and major connections in X0 are designed to allow the device to be split into a pre-processing section on the left, and a post processing section on the right half of the FPGA.

Interface

The SLAAC interface includes a Xilinx XC4062XLA-09 and several supporting components for clock generation and distribution, configuration, power management, external memory access, and system bus interfacing.

Clock. The SLAAC interface includes a clock generator tunable from 391 KHz to 100 MHz increments of 1 MHz. Clock distribution is separated into two domains. A processor clock (PCLK) drives the logic in X0, X1, and X2. PCLK is looped through the interface FPGA to support flexible countdown timers and single-step clocking. A memory clock (MCLK) drives the user memories and allows the host to access the memories while the PCLK is halted.

External Memory Bus. All of the user programmable memories in the SLAAC architecture are accessible from the host through an external memory bus. This feature guarantees a stable path to the memories for initialization, debugging, and retrieving results without depending upon the state of the user FPGAs. For each memory, a pair of transceivers isolates the address/control and data lines from the shared external memory bus. The transceivers are controlled from the IF chip.

For the SLAAC-1 architecture, we chose to implement a preemptive memory access strategy similar to that of Splash 2. In a preemptive memory access, the host interrupts the user FPGAs to read or write the memory. The user FPGAs are unaware that the access has occurred. This greatly simplifies the user's design because exclusive access to the memory is assumed. No special states are required in user state machines for initialization or debugging.

Although we chose a preemptive memory access strategy for SLAAC-1 for our interface implementation, the fact that the interface is within an FPGA allows us to explore other approaches. In addition to this, the fact that the SLAAC-1 memories and transceivers that implement the external memory bus are located on replaceable memory modules (see Figure 3) presents ample opportunity to experiment with alternate memory designs. Each of the memory modules has 160 undedicated pins connected to one of the processing elements and a 40-pin connection to the external memory bus.

Configuration. The IF device is programmed on power-up by an EEPROM to provide a stable interface to the host. The EEPROM program pins are accessible to the host through a control/status register in IF. This enables in-system updates of the interface through software. The user programmable FPGAs in the system are configured from IF. X0, X1, and X2 can be programmed individually or in parallel. A simple slave bus configuration through a set of control/status registers is supported for the SLAAC-1 prototype. However, there are two additional memories on the external memory bus dedicated to the IF to act as a configuration cache. The host can quickly load the configuration cache and the configuration can occur autonomously in IF, thus freeing up the host more quickly. An added benefit of placing the configuration memories on the external memory bus is that any or all of the ten user memories can be conscripted as configuration caches. Up to six complete SLAAC-1 configurations (including X0, X1, and X2) can be stored simultaneously on SLAAC-1 and selected with minimal effort from the host.

Readback. An integral part of rapid prototyping on reconfigurable architectures is the ability to debug a design on the hardware. The Xilinx readback facility is essential. The IF chip provides readback access to X0, X1, and X2 through a set of control/status registers. For the SLAAC-1 prototype, a simple slave bus readback is supported. However, the configuration cache memories can also be used as a readback cache.

FIFOs. Instead of dedicated hardware, a number of input and output FIFOs are implemented within the logic of the interface chip. The SLAAC-1 architecture is designed to allow the user FPGA logic to simultaneously process a number of input and output streams. This feature is essential for the network-centric SLAAC system architecture. The ability to address multiple input and output FIFOs allows the user FPGAs to dynamically route data across multiple network channels on a cycle-by-cycle basis [2].

Power Management. Since the user logic in X0, X1, and X2 has the potential of drawing too much current for the PCI slot, the SLAAC interface includes a power monitoring circuit. Power monitoring is accomplished using a current to voltage monitoring circuit on the +5V, +3.3V and +2.5V supply lines. Each circuit uses a LMC6482 operational amplifier and a low value current sensing resistor. Feedback resistors set the appropriate gain. These analog voltage levels are then monitored by a PIC16715E microcontroller that has four A/D input channels available [6]. Once a threshold level has been triggered the PIC interrupts the IF device. The IF design is able to halt the processor clock to stop the user FPGAs and interrupt the host.

IMPLEMENTATIONS AND STATUS

SLAAC-1 (PCI)

SLAAC-1 is a full-sized PCI board designed for use in the RRP workstations. Although the initial release of the interface FPGA contains a Xilinx 32-bit PCI core, the hardware is capable of supporting 64-bit PCI. Figure 2 contains a photograph of SLAAC-1 assembled in March 1999. In order from left to right, the large BGA devices are IF, X0, X2, and X1. The double-row of 100-pin connectors above X0, X1, and X2 support memory daughter card modules.

A memory module is show in Figure 3. Each memory module has four 256Kx18 synchronous SRAMs and the transceivers for the external memory bus. The memory module for the IF and X0 devices share one memory card since there are two memories on X0 and two configuration cache memories on IF.

On the back of the SLAAC-1 board (not shown) are four systolic connectors for the high-speed data path through the X1 and X2 chips. The 64 data bits of the X0 to X1 and the X2 to X0 ring paths are shared with the systolic connectors. Additional pins from the X1 and X2 chips provide control for the respective external data sources.

SLAAC-2 (VME)

SLAAC-2 is a 6U VME mezzanine board designed to plug into a modified CSPI M2641 baseboard carrier. As shown in the SLAAC-2 architecture diagram in Figure 4, there are actually two SLAAC-1 compatible accelerators on the SLAAC-2 board, each controlled by one of the two PowerPCs on the M2641. A few modifications were necessary to the basic SLAAC-1 design to accommodate having two accelerators in an area not much larger than a single full-sized PCI board. However, most of these changes are not directly visible to the SLAAC-2 application designer.

One change to the SLAAC-2 design was that the Xilinx 4062 IF device on SLAAC-1 was replaced with Xilinx 4085s. The extra I/O pins available on the 4085 were needed to accommodate the unmultiplexed 64-bit PowerPC bus. Other modifications were made to save space on the board, including combining the power management, IF boot EEPROMS, and the reference oscillator. The external memory bus was the only casualty to compute density visible to the user. There was insufficient area available for the transceivers necessary to isolate the external memory bus during normal compute FPGA operation. It was decided that since the SLAAC-1 and SLAAC-2 user FPGAs are bitfile compatible, debugging the memories could happen on a PCI board and was not essential in the VME platform. The only consideration for the application designer is that the memories will have to be loaded from within X0, X1, and X2.

Also shown in the SLAAC-2 architecture diagram are two 40-pin busses between X1A and X2B, and X1B and X2A. The spare pins used for controlling the external systolic connectors in SLAAC-1 were used in SLAAC-2 to bridge the two “independent” designs. Although the A and B designs have separate tunable clock synthesizers, a side-effect of having a single reference oscillator on SLAAC-2 we believe will allow the two designs to operate synchronously with each other. In any event, we cross-clocked to spare pins in the compute FPGAs so that X1A and X2A have access to the B design’s clock and vice versa with X2A and X2B. This permits cooperation between the two adjacent nodes.

Figure 5 is a photo of the component side of SLAAC-2. A total of six FPGAs is visible on this side. Two additional FPGAs on the back are not shown. The long connectors visible nearly spanning the length of the board are two PowerPC bus connectors.

M2641S

The commercial CSPI M2641™ has four 300MHz PowerPC 603r processors connected by a Myrinet 1.2Gb/sec SAN network. The M2641 has an integrated 8-port Myrinet switch and supports network connections from both the front-panel and the P0 VME backplane row for cable-free networking [7]. Figure 6 shows the M2641S carrier that has been modified for SLAAC-2. The black circular heat-syncs conceal the Power PC processors in the upper-left and upper-right corners. The BGA packages near the center of the board from left to right are 1) an ASIC interfacing the 2) Myricom LANai network processor to the PowerPC bus, and the 3) LANai processor and 4) ASIC FPGA interface for the second PowerPC node. The M2641S board has been modified for the SLAAC project to include two 120-pin PowerPC bus connectors visible in the SLAAC-2 photo. Two additional 60-pin SAN connectors are also available. However, they are unpopulated in this photograph because SLAAC-2 does not use them.

APPLICATIONS

As a validation of the SLAAC architecture approach, we are implementing portions of the Joint STARS Synthetic Aperture Radar / Automatic Target Recognition (SAR/ATR) application from Sandia National Labs on the SLAAC-2 system. The three primary components of the Sandia SAR/ATR application are Focus-Of-Attention (FOA), Second-Level Detection (SLD), and Final Identification (FI). SLD is the most computationally intensive of the three algorithms used in Sandia SAR/ATR application [8]. In SLD, regions of interest, called *chips*, produced by the FOA algorithm are correlated with target templates. Figure 7 shows the SAR/ATR algorithm data flow.

The SLD algorithm can be described mathematically as follows. Two hundred sixteen template pairs represent each target type from 72 orientations (rotations) and 3 angles of elevation. A template pair comprises a bright template, which represents pixels with a strong radar return, and a surround template, which represents pixels with strong radar absorption. The chip image is contained in the 64x64-pixel matrix *M*. The 32x32-pixel bright and surround templates are contained in the matrices *B* and *S*. Let *Bias* be a template-specific value used to set the adaptive threshold. For each position *(i, j)* in the search space, SLD can be computed in five phases (*P1 – P5*):

$$P1: SM(i,j) = \sum_{u=0}^{31} \sum_{v=0}^{31} B(u,v) M(i+u, j+v)$$
$$P2: TH(i,j) = \frac{SM(i,j)}{BC} - Bias$$
$$P3: BS(i,j) = \sum_{u=0}^{31} \sum_{v=0}^{31} B(u,v)[M(i+u, j+v) \geq TH(i,j)]$$
$$P4: SS(i,j) = \sum_{u=0}^{31} \sum_{v=0}^{31} S(u,v)[M(i+u, j+v) < TH(i,j)]$$
$$P5: Q(i,j) = \frac{1}{2}\left(\frac{BS(i,j)}{BC} + \frac{SS(i,j)}{SC} \right)$$

Where a hit at position *(i, j)* is valid if:

$$TH(i,j) \leq TH_{\max}$$
$$TH(i,j) \geq TH_{\min}$$
$$BS(i,j) \geq BS_{\min}$$
$$SS(i,j) \geq SS_{\min}$$

And the variables are defined as shown in Table 1. SLD returns the two highest quality hits for each chip.

The search space is defined by the set of pixels in the chip that are used as a correlation operation origin. Graphically, the origin is the lower left corner of the correlation images. The origins that are used in SLD are those pixels in the chip where the lower left of the template can be overlaid without any of the template going outside the chip. The current version of FOA guarantees that the no target pixels are within nine pixels of the edge of the chip, reducing the effective chip size to 46x46. Since the template size is 32x32, the size of the search space is 15x15 (46-32+1=15).

Our SLAAC-2 implementation of the SLD algorithm is based on an FPGA mapping created by Myricom, Inc. [9]. In our implementation, we store the target templates in the SRAMs local to the compute FPGAs, X1 and X2. The image chips are broadcast from the host, through IF and X0, to the compute elements in X1 and X2. Each compute element in X1 and X2 computes a correlation for a relative placement for a single chip and template pair. The thresholds, surround sum, and bright sums are passed through X0, which does the comparisons to determine whether a hit was found, and passes the hits back to the general-purpose processor. The general purpose processor determines the *k* best hits, where *k* can be determined at runtime. Determining the hits on X0 allows the number of FPGAs per general purpose processor to be scaled, while allowing the *k* best hits on the general-purpose processor.

Our estimates are that each compute element will process pixels at 40 MHz, with approximately 75 compute elements per compute FPGA. With two compute FPGAs per SLAAC board, this brute force method will achieve approximately 15,000 template matches per second. With optimizations, we believe we can achieve 30,000 template matches per second by skipping zero elements in the templates. A computation rate of 30,000 template matches per second per VME slot is three times the performance achieved by Myricom's implementation, and is approximately four times faster than the estimated performance of a quad-PowerPC board. As FPGAs get larger, we expect performance of SLD at least linearly. The parallelism available in the algorithm allows increased density of devices to translate directly into increased performance in terms of the number of computational elements per chip and clock speed. For example, we would expect a SLAAC board populated with Xilinx Virtex XV1000 parts, with 1,000,000 gates per chip, to achieve at least 200,000 templates/second, twenty times the performance of a quad-PowerPC board. Furthermore, architectural improvements which will allow better arithmetic implementations will further increases in functional density and clock speed.

FUTURE WORK

The SLAAC team has an aggressive schedule of demonstrations in the coming year on both the SLAAC-1 and SLAAC-2 systems. Some applications include IR/ATR, SAR/ATR, Sonar Beamforming, and Multi-dimensional image processing. Incremental releases are planned to improve the performance of the interface and device drivers. Other future work includes supporting JHDL design environment and extending the SLAAC VHDL simulator to include multiple-board systems.

ACKNOWLEDGEMENTS

We would like to thank Jose Munoz, Bob Parker, Jeff Arnold, Peter Athanas, Brian Bray, John Granacki, Brad Hutchings, Mark Jones, Jeff LaCoss, Kevin McCabe, and Brent Nelson for contributions to the SLAAC-1 and SLAAC-2 designs.

This research is sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory, or the U.S. Government.

REFERENCES

[1] D. Ridge, D. Becker, P. Merkey, and T. Sterling, "*Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs,*" in Proceedings, IEEE Aerospace, 1997.

[2] P. Athanas, M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, "*Implementing an API for Distributed Adaptive Computing Systems*," submitted to the *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines,* April 1999.

[3] D. A. Buell, J. M. Arnold and W. J. Kleinfelder, "*Splash 2 FPGAs in a Custom Computing Machine,*" IEEE CS Press, Los Alamitos, CA, 1996.

[4] Annapolis Microsystems Incorporated, *“WildForce User’s Guide,” 1998.*

[5] Micron Technology, Inc., *“Micron MT255L256V18F ZBTÔ SRAM Data Sheet”*, 1999 [http://www.micron.com/mti/msp/pdf/datasheets/3702.pdf].

[6] Microchip Technology, Inc., *“PICmicro™ Devices,”* 1999 [http://www.microchip.com/10/Lit/PICmicro/index.htm].

[7] CSPI, Inc., *“M2641 Multicomputer Datasheet,”* 1999 [http://www.cspi.com/multicomputer/index.htm].

[8] M. Rencher, B. Hutchings, "*Automated Target Recognition on SPLASH 2*", in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997.

[9] W-K. Su, R. Sivilotti, Y. Cho, and D. Cohen, “*Scalable, Network-Connected Reconfigurable, Hardware Accelerators for an Automatic-Target-Recognition Application*,” Web page, http://www.myri.com/research/darpa/atr-report.pdf. May 1998.

FIGURES

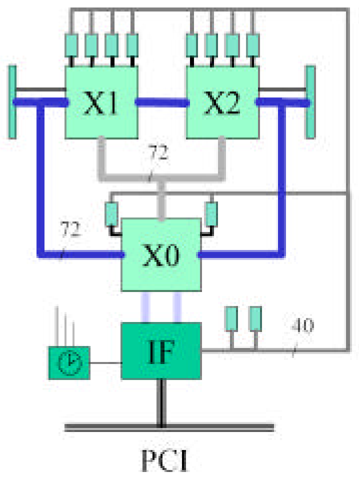


Figure 1. SLAAC-1 Architecture Diagram

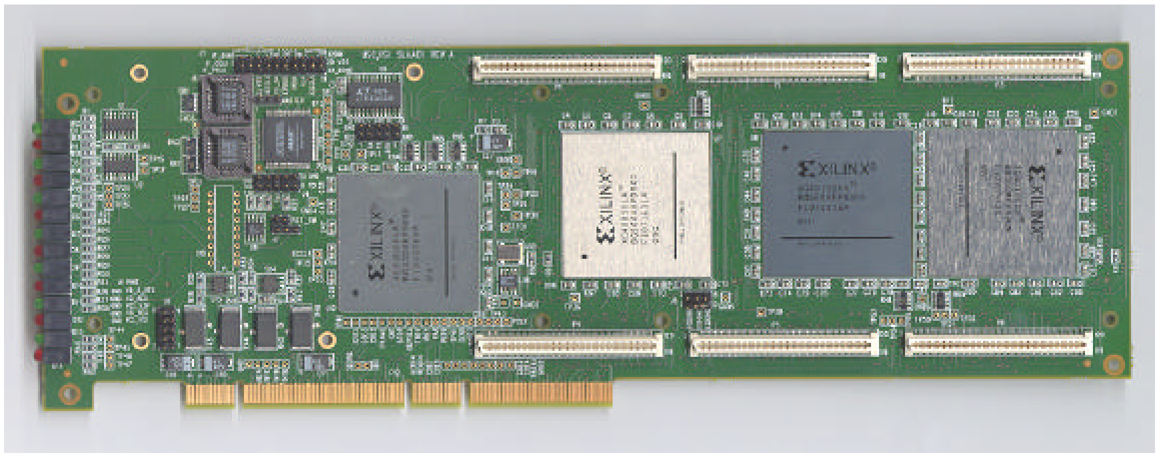


Figure 2. SLAAC-1 Photo

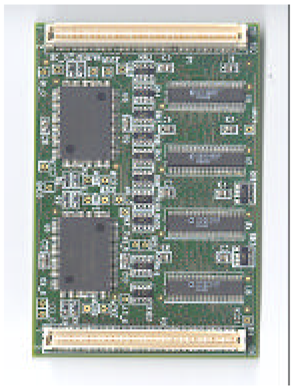


Figure 3. SLAAC-1 Memory Module

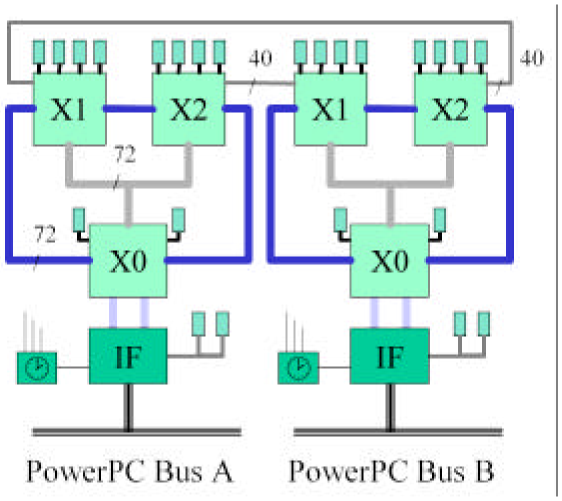


Figure 4. SLAAC-2 Architecture Diagram

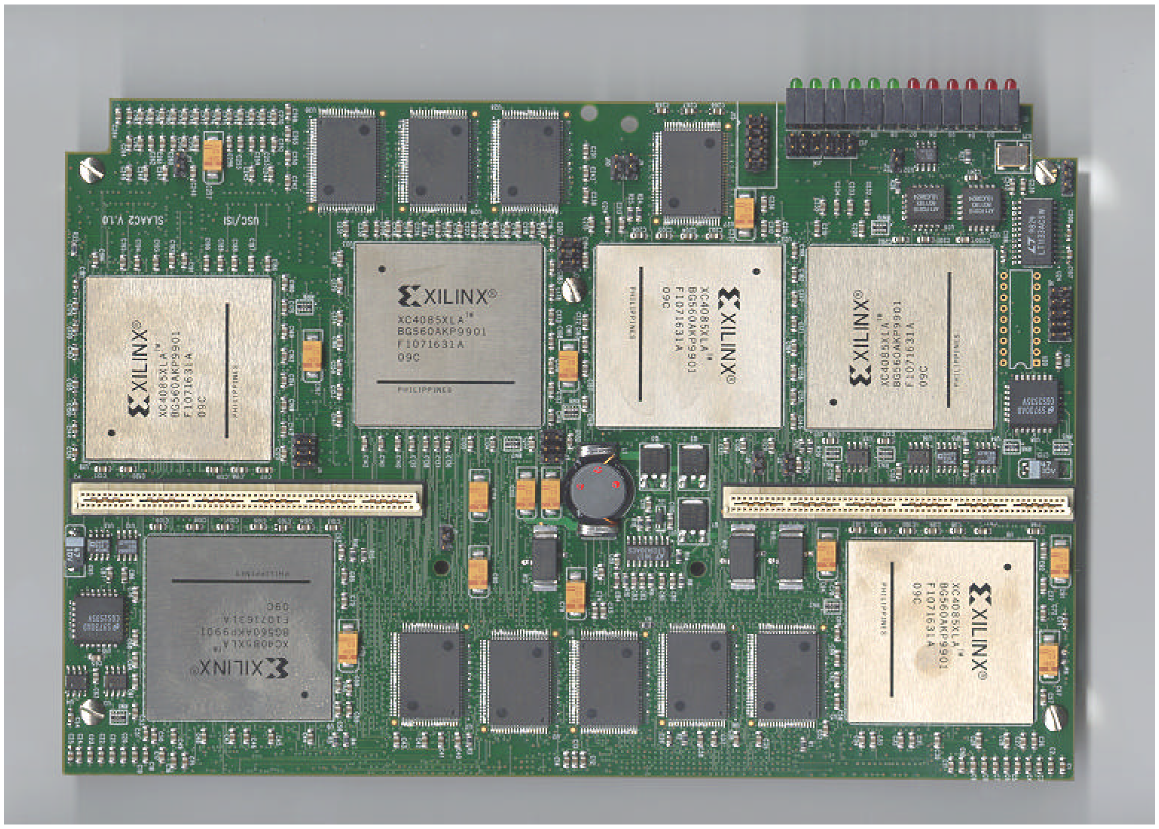


Figure 5. SLAAC-2 Photo

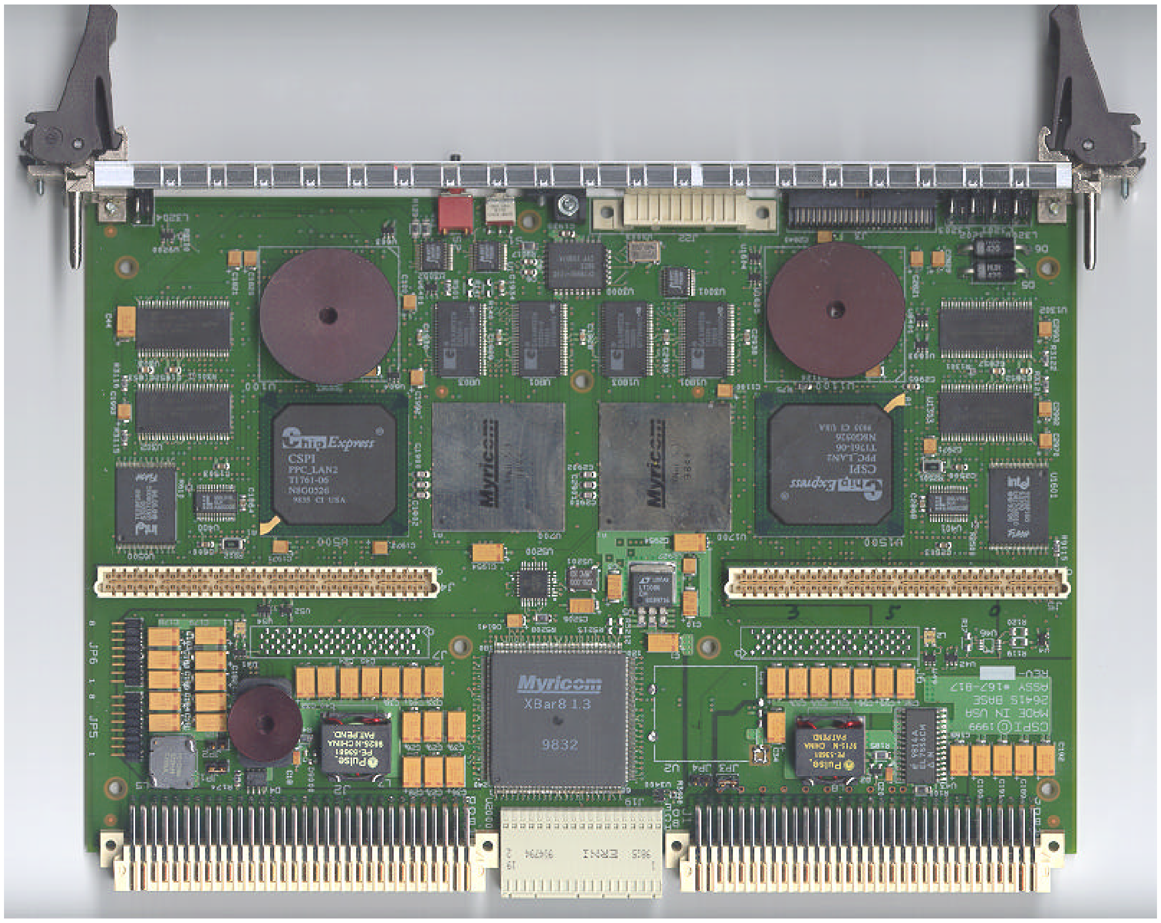


Figure 6. M2641S Photo

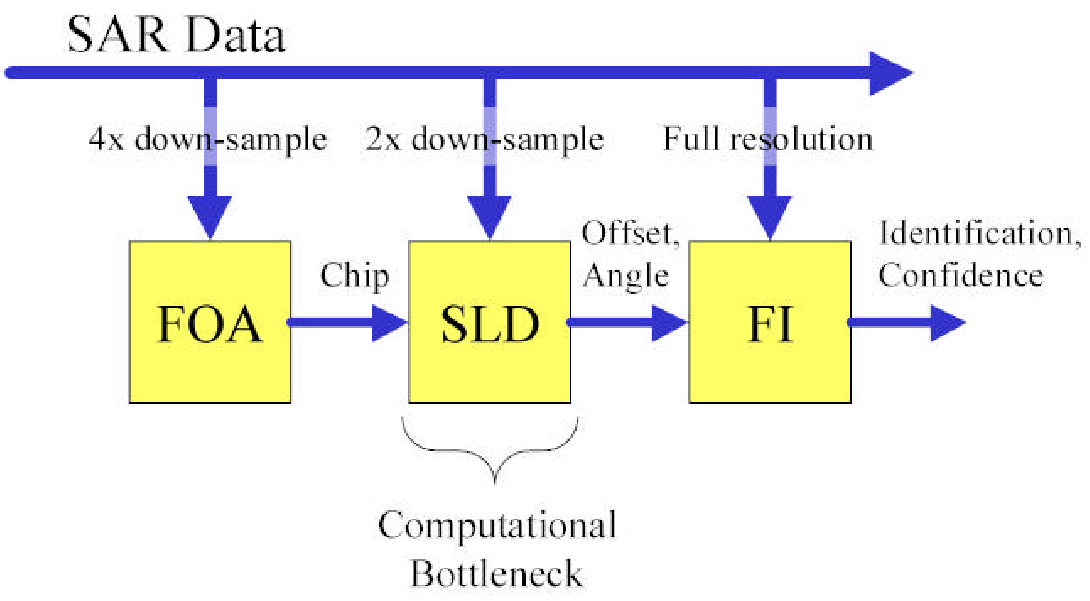


Figure 7. SAR/ATR Algorithm Flow

Variable	**Description**	**Bit Width**
B	Bright template	1
S	Surround template	1
M	Chip (image)	8
BC	Number of ones in bright template	10
Bias	Template-specific intensity bias	8
SM	Shape sum	16
TH	Threshold	8
BS	Bright sum	8
SS	Surround sum	8
Q	Hit Quality	8

Table 1. SLD Equation Variable Definitions

BIOGRAPHIES

Brian Schott is a systems programmer in the Adaptive Computing Systems lab at the University of Southern California, Information Sciences Institute. He is currently a member of the SLAAC project. Prior to working at ISI, Mr. Schott was a member of the SPLASH 2 team at the Center for Computing Sciences (formerly called the Supercomputing Research Center) where he gained considerable experience programming reconfigurable logic-based machines. In addition to other projects at CCS, Mr. Schott worked on the dbC compiler for Splash 2, a SIMD C with bit-level extensions. The dbC compiler generates the program and a custom SIMD instruction set for an application and a custom SIMD machine is automatically synthesized for Splash 2. Mr. Schott received his BSCS at the University of Maryland and is currently pursuing his MSCS at George Washington University.

Dr. Stephen Crago is a Computer Scientist in the Adaptive Computing Systems lab at the University of Southern California's Information Sciences Institute. He completed his Ph.D. in Computer Engineering from the University of Southern California in 1998, where he developed the HiDISC architecture, a microprocessor architecture designed to tolerate memory latency. He completed his MSEE and BSCEE in 1990 and 1992, both at Purdue University. He has been a Computer Scientist since 1997 at ISI East, where he has worked on adaptive computing architectures, SAR/ATR algorithm implementations, and embedded parallel computing for space.

Reconfigurable Processors for High-Performance, Embedded Digital Signal Processing[⋆]

Paul Graham and Brent Nelson

459 Clyde Building, Brigham Young University, Provo UT 84602, USA
`grahamp@ee.byu.edu, nelson@ee.byu.edu`

Abstract. For high-performance, embedded digital signal processing, digital signal processors (DSPs) are very important. Further, they have many features which make their integration with on-chip reconfigurable logic (RL) resources feasible and beneficial. In this paper, we discuss how this integration might be done and the potential area costs and performance benefits of incorporating RL onto a DSP chip. For our proposed architecture, a reconfigurable coprocessor can provide speed-ups ranging from 2-32x with an area cost of about a second DSP core for a set of signal processing applications and kernels.

1 Introduction

For high-performance, embedded digital signal processing, digital signal processors (DSPs) are very important, but, in some cases, DSPs alone cannot provide adequate amounts of computational power. As shown in [1, 2], reconfigurable logic (RL), specifically, FPGAs, can profitably be used for signal processing applications which require large amounts of computation and outperform existing high-performance DSPs despite the weaknesses of current commercial FPGAs in performing arithmetic. Unfortunately, FPGAs cannot effectively handle applications requiring high-precision arithmetic or complex control. As a compromise, we have been exploring DSP-RL hybrid architectures which enjoy the flexibility and precision of DSPs while experiencing performance improvements due to RL.

In this paper, we introduce a hybrid DSP-RL processor which tightly couples the DSP core to reconfigurable logic for greater performance and flexibility with digital signal processing applications without altering the DSP's embeddable nature. First, we will discuss the benefits and drawbacks of such an architecture. Following this discussion, we will describe many of the possible hybrid architectures and provide our performance and silicon-area estimates for one specific architecture. The conclusion to the paper provides some summary comments on this hybrid and a few words regarding on-going and future work in this area.

⋆ Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

2 Why a DSP Hybrid?

A large number of projects, including [3, 4, 5, 6, 7, 8], have explored the use of processor-RL hybrids for "general-purpose" computing applications. These projects have generally involved coupling a RISC, often a MIPS, processor core with reconfigurable logic and, by doing so, have shown promising speed-ups for a range of applications, including DES encryption, the SPECint92 benchmarks, and image processing. The combination of the processor core with either reconfigurable function units or coprocessors enables the host processor to communicate through high-bandwidth, on-chip interconnection to the RL resources rather than the relatively slow interconnection made through an interface with an external system bus. With reconfigurable resources, these processors were able to compute with greater efficiency than software alone, having less software overhead due to address generation, branching, and function calls and exploiting more parallelism than is possible with the processor's normal data path.

One hybrid processor-RL system called Pleiades [9, 10] has specifically targeted ultra-low power, embedded digital signal processing. The approach fuses an ARM core with a combination of reconfigurable logic, reconfigurable data path, and/or reconfigurable data-flow resources. In this case, due to power and performance constraints, the RISC core is used mainly for administrative tasks such as programming the reconfigurable resources and not for computation. According to the results in the above-cited papers, the architecture has proven its power-performance advantage over a number of common technologies such as DSPs and FPGAs for a few digital signal processing applications.

Unfortunately, RISC processors, despite their computation power, are not well suited for many embedded high-performance applications. These processors are often power hungry, require a number of support ICs, and exhibit behavior which is hard to deterministically characterize for real-time applications. These qualities often make DSPs better choices for embedded applications. Further, a handful of DSPs offer glueless multiprocessing and provide a computing density per board advantage over multiple embedded RISC processors—a metric very important to embedded system designers.

Though DSPs may be good candidates for embedded processors, they also have some disadvantages resulting from their design as efficient embeddable processors. For instance, programming DSPs is laborious relative to programming RISC processors since the related tools are less sophisticated and less efficient. Much of this follows from the emphasis on making DSPs both power and memory efficient as opposed to being compiler friendly. Further, DSPs' explicit parallelism must be managed efficiently, which is not always an easy task. Also, due to the emphasis on low-power and low-latency design, DSPs tend to operate at lower clock frequencies than RISC processors.

Despite these drawbacks, DSP cores have several existing features which make them good candidates for supporting reconfigurable resources. First, many DSPs have either multiple on-chip memory banks or otherwise provide multiple memory accesses per cycle. Thus, the memory architecture of the processor does not have to be drastically modified to provide many independent memory ports to the reconfigurable resources. Second, the parallel execution of the DSP core and the RL resources can often be expressed as simple extensions of DSP instruction sets, which already directly express some level of concurrency. Third, the on-chip DMA controllers which many DSPs already have may prove useful for configuring the RL without burdening the processor with the task. Moreover, since the execution of many DSP applications are quite deterministic, configuration pre-fetching scheduled at compile time [11] should be a useful technique. The expense of configuration management hardware does not appear to be justified for a DSP-RL hybrid processor if only a few kernels are deterministically loaded during application execution.

DSPs can benefit in several ways from using reconfigurable logic for on-chip processing. First, DSPs are often asked to perform tasks which require a large amount of bit-level data manipulation such as error control coding—functions for which they are ill-suited but can perform. Second, the parallelism which is inherent in many DSP applications often cannot be exploited well by a processor which can, at most, perform a few arithmetic operations per cycle. Some recently announced DSPs such as the Analog Devices' Hammerhead and Tiger SHARCs use SIMD techniques to boost the performance of the processors for some applications, but, unfortunately, these techniques can only be used for a portion of all DSP applications. The parallelism that the on-chip reconfigurable resources can exploit is not limited to SIMD techniques. Another interesting, but minor benefit that DSPs can experience by using RL resources on-chip is the ability to generate application-specific address streams with RL. In [1], we discussed an application, delay-sum sonar beamforming, which required address generation for which the usual DSP hardware address generators were inefficient. As we show in Sect. 4, RL used just for address generation can provide a three-times speed-up for this application's kernel at a very small silicon area cost. Lastly, RL can lead to more efficient use of memory bandwidth by performing computations on data as they are streamed to and from memory.

The DSP core also provides the hybrid processor with capabilities which a strictly reconfigurable architecture cannot offer. First, due to the fast reconfigurability of the DSP core from instruction to instruction and the size of its instruction store, the processor is better suited for complex, control-heavy portions of applications. Second, depending on the processor core used, the core can provide higher precision arithmetic than may be practical in the RL resources, meaning that the processor core and RL can play complementary roles in applications. The processor core can deal with operations which require high precision while the RL can provide speed-ups for functions which require large amounts of computation on lower-precision data objects.

3 Hybrid DSP Architectures

For our architectural studies, we have decided to use the Analog Devices' SHARC DSP family as a model because of its features and performance in DSP applica-

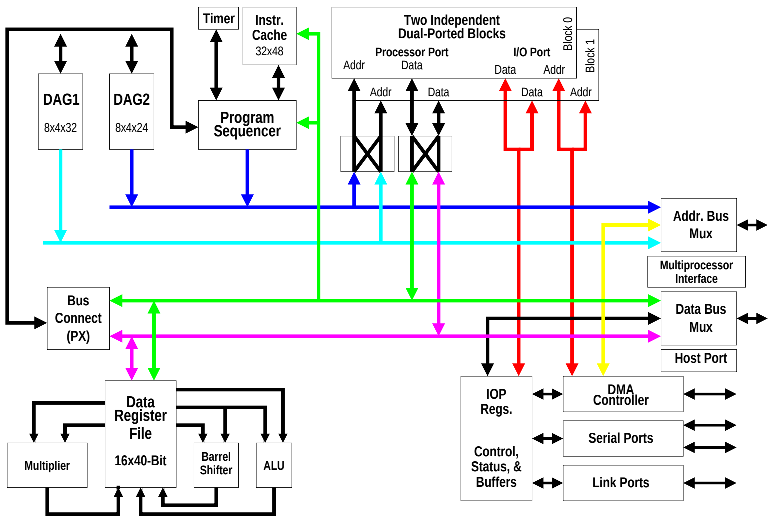


Fig. 1. SHARC Architecture

tions [12]. A block diagram of the architecture is provided in Fig. 1. Among the features which influenced our choice of the SHARC are:

1. its large on-chip memory (512 KB) organized as two banks of dual-ported memory, allowing for concurrent off-chip I/O and processing;
2. a data path capable of performing up to 3 arithmetic operations per cycle (a multiply with subtract and add);
3. several on-chip I/O processor peripherals including DMA controllers;
4. and, the support of glueless multiprocessing.

We chose this DSP despite the fact that it was a floating-point DSP and the applications and kernels in the benchmarks, described in Sect. 4, are implemented in fixed-point. Our rationale for this choice was that no fixed-point DSP currently has features equivalent to those listed above, all of which greatly improve the processor's performance for large applications. Second, considering that the DSP core is only 12% of the total die area, we do not believe the area comparisons would be greatly affected if the core was a fixed-point processor.

The reconfigurable portion of a DSP-RL architecture can be interfaced with the DSP core in several ways. First, the reconfigurable logic can act much like a function unit in the data path of the processor, having access only to the processor's register file. This approach is problematic since the memory bandwidth to the reconfigurable logic is effectively restricted to the one or two memory ports which service the register file. As was shown in [1, 13] and other work, restricting memory ports to only one or two greatly limits the performance of the architecture and causes the memory subsystem to be the bottleneck.

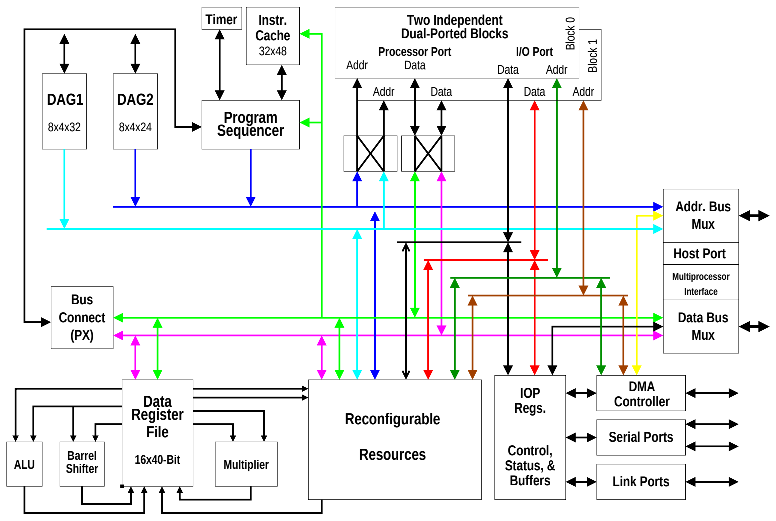


Fig. 2. DSP with a Reconfigurable FU/Coprocessor Combination

Another alternative is to treat the reconfigurable portion of the processor as a coprocessor. If the reconfigurable logic is treated as another peripheral such as the I/O processor of the SHARC, it again has only two memory ports and must compete with the I/O processor for memory, but the RL can operate concurrently with the processor core without disturbing the core's operation. Of course, this does not alleviate the lack of memory ports.

Another variation of this architecture is to modify the memory interfaces to provide a total of 4 memory ports for the reconfigurable processor; in the SHARC architecture, this means sharing both the DSP core's memory ports and the I/O processors' memory ports, i.e., having access to all of the memory ports of the two dual-ported, on-chip memories. Though the DSP cannot run concurrently with the reconfigurable coprocessor when all of the memory ports are being used by the coprocessor, the accessibility of the independent memory ports can enable the reconfigurable coprocessor to speed up applications beyond what the DSP core itself can provide. With the proper, careful assignment of memory ports, concurrent operation should still be possible for the DSP and reconfigurable coprocessor with this structure.

Among the other possibilities, another variation is to provide the reconfigurable resources both access to the processor register file and the four on-chip memory ports; this is illustrated in Fig. 2. This makes the reconfigurable resources a cross between a function unit and a coprocessor. This approach provides flexibility and reduces the requirement of using memory as the means of communication between the two data paths, further reducing the memory bandwidth burden and allowing tighter cooperation between the DSP core and the RL.

Table 1. Benchmark Applications and Kernels

Function	Description
BYU Applications	
Delay-Sum Beamforming	Beamforming in which the phase shifts are performed by time delays
Frequency-Domain Beamforming	Beamforming in which the phase shifts are performed in the frequency domain
Matched Field Beamforming	Beamforming in which a multi-ray model is used to estimate angle and distance of the target.
Benchmarks Inspired by BDTImark benchmark	
Real Block FIR	Finite impulse response filter that operates on a block of real (not complex) data.
Complex Block FIR	FIR filter that operates on a block of complex data.
Real Single-Sample FIR	FIR filter that operates on a single sample of real data.
LMS Adaptive FIR	Least-mean-square adaptive filter; operates on a single sample of real data.
IIR	Infinite impulse response filter that operates on a single sample of real data.
Vector Dot Product	Sum of the point-wise multiplication of two vectors.
Vector Add	Point-wise addition of two vectors, producing a third vector.
Vector Maximum	Find the value and location of the maximum value in a vector.
Convolutional Encoder	Apply convolutional forward error correction code to a block of bits.
Finite State Machine	A contrived series of control (test, branch, push, pop) and bit manipulation operations.
256-Point, Radix-2, In-Place FFT	Fast Fourier transform converts a normal time-domain signal to the frequency domain.

4 Area and Performance Results

As a starting point for our work, we have been evaluating the architecture of Fig. 2 for performance potential and silicon area considerations. As a way of benchmarking the performance, we have chosen a collection of applications and kernels which represent typical digital signal processing computations; these are listed in Table 1. In the mix of applications and kernels, we have included a few sonar beamforming applications [1, 14] with which we are very familiar. The other kernels are modeled after the collection of kernels in the BDTImark benchmark defined by Berkeley Design Technology Inc., who have identified these kernels as important to many DSP applications[15].

Our initial studies assume that the reconfigurable logic looks much like the Xilinx 4000 family of FPGAs. Though we expect coarser-grain RL architectures such as CHESS [16], CFPA [17], and the Garp RL array [6] would be better for the hybrid, we use this RL architecture because it is well known and understood. Also, it provides a nice upper bound for the area-performance trade-offs in the hybrid architecture since the Xilinx 4000 architecture will generally be more costly for implementing data-path operations than the coarse-grain field programmable architectures mentioned above. In fact, we expect these coarse-grain architectures to require only about 40% to 55% of the silicon area of the Xilinx 4000 architecture for the same data-path functionality. To its credit, though, the Xilinx 4000 architecture will often be more flexible for control hardware and bit-level operations due to its finer granularity.

From [12], we learn that the SHARC, using a .6 micron, two-layer metal process, requires about $3.32x10^9\,\lambda^2$ in area. Due to the large on-chip memories of the architecture, about $1.86x10^9\,\lambda^2$, or 56% of the die, is devoted to SRAM, while about 12% of the die area, or $3.99x10^8\,\lambda^2$, is the DSP core and data path. The remaining die area is devoted to on-chip peripherals, the I/O pads, and interfacing logic. With this information and area estimates for Xilinx 4000 CLBs with interconnect drawn from [18], we can estimate the relative area increases due to the reconfigurable logic for each of the applications in the benchmark suite. For instance, from [18], we learn that a CLB has a cost of approximately $1.26x10^6\,\lambda^2$, so a design requiring 100 CLBs would require about $1.26x10^8\,\lambda^2$ in silicon area, which is only a 3.79% increase in total die area.

Table 2 describes the performance per area increase for the hybrid architecture for several of these benchmarks. The entries are sorted in decreasing speed-up per silicon area change. As an explanatory note, we should point out that the speed-up numbers do not take into account reconfiguration time for the kernel or application because our analysis assumes that reconfiguration is infrequent. Also, the RL implementations account for bit growth due to arithmetic operations, though simple scaling is essential in some cases. You should further note that only the reconfigurable logic portion of the hybrid is executing the benchmarks, except in a few cases that are noted in the table where the RL and the DSP core are operating cooperatively. In this study, we also assume that the reconfigurable coprocessor operates at the same clock frequency as the DSP. Though, in general, we would not expect this from the Xilinx 4000 architecture for higher-frequency DSPs (100–200 MHz), we expect that coarser-grain, data-path-oriented field-programmable architectures such as CHESS, CFPA, or the Garp RL array would be able to support these frequencies. For instance, the Garp RL array has been estimated to execute at a minimum of 133 MHz[6]. Lastly, as mentioned before, the reconfigurable implementations of the benchmarks are fixed-point implementations, not floating point.

With a reconfigurable logic budget of about 1000 CLBs, a good portion of the kernels and applications can be accelerated without frequent reconfiguration of the RL. This accounts for about a 40% increase in chip area. From our estimates and the published work on other field-programmable architectures, we believe that the cost of this amount of reconfigurable logic can be as little as 16–20% with computational fabrics such as CHESS or CFPA—an area just a

Table 2. Performance and Area of Hybrid for Benchmark Circuits

Application	Implementation	Pipe-lining	Speed-Up	Area (CLBs)	Total Area	Speed-Up / Δ Area
Convolutional Encoder	For V.32 Modem, with differential encoding	Full	34	5	1.0019	17929
Vector Add	Cooperative w/ DSP	N/A	2	40	1.0152	131
Vector Max.	16b data	Full	2	48	1.0182	110
Delay Sum BF	Coop. w/ DSP (Addr. Gen.)	N/A	3	100	1.0379	79.1
Delay Sum	Full PE	Full	3	246	1.0933	32.2
IIR Biquad	4 8b $\times$, 4 20b +	1/2	4	360	1.1365	29.3
FIR	32 tap, 16b KCM, 40b +	1/2	32	2976	2.1287	28.4
FIR	16 tap, 16b KCM, 40b +	1/2	16	1488	1.5643	28.4
FIR	2 tap, 16b KCM, 40b +	1/2	2	186	1.0705	28.4
FFT	CORDIC-based butterfly	1/2	4	380	1.1441	27.8
Matched Field BF	Sub-voxel, memory sensitive	1/2	8	928	1.3520	22.7
Matched Field BF	Sub-voxel, memory sensitive	Full	8	1073	1.4070	19.7
Vector Dot	Coop. w/ DSP, 16b $\times$, 40b +	1/2	2	308	1.1168	17.1
Matched Field BF	Sub-voxel, memory intensive	1/2	6	928	1.3520	17.0
Matched Field BF	Sub-voxel, memory intensive	Full	6	1073	1.4070	14.7
Freq. BF	Reported in [1]	1/2	4	856	1.3247	12.3
FIR	2 tap, 16b $\times$, 40b +	1/2	2	552	1.2094	9.55
IIR Biquad	4 16b $\times$, 4 24b +	1/2	4	1200	1.4551	8.79
IIR Biquad	5 16b $\times$, 4 24b +	1/2	4	1488	1.5643	7.09

little larger than the size of the DSP core. In addition, since the computation density per board is often one of the most important characteristics for embedded system designers, silicon area increases of 16%–40% are acceptable if the total computational density per board is increased by a factor of 2 or more.

5 Conclusions and Future Work

We have demonstrated that a reconfigurable architecture which incorporates a reconfigurable coprocessor into a DSP can have performance benefits for a reasonable increase in chip area. In addition, DSPs have many architectural features which make a combination with reconfigurable logic feasible and beneficial. Despite the raw clock rate disadvantage of DSPs when compared with general-purpose microprocessors, DSPs serve an important role in high-performance, embedded computing; a reconfigurable coprocessor on-chip can help DSPs exploit more of the parallelism found in digital signal processing applications, thus improving the processor's overall performance.

A large amount of work still remains to be performed and many outstanding questions exist. For instance, the amount of concurrent DSP-RL execution which can be expected for DSP algorithms should be determined—a process which may require the use of automated programming tools and the mapping of many DSP algorithms to the architecture. If our current experience is representative, there may not be many situations in which concurrent operation is beneficial or possible, indicating that the connection of the RL array to the DSP's register file may not be needed.

Another unanswered question is whether the DSP and RL can truly serve complementary roles in applications—the DSP performing the control-heavy and high-precision arithmetic portions of the algorithm while the RL performs highly parallel operations on lower-precision data items. Again, a large amount of application mapping to the architecture may be required to answer this question, though adaptive signal processing algorithms may provide a fruitful set of applications for this study.

Other on-going and future work include a refinement of the programming methodology for the DSP hybrid processor as well as the many issues of interfacing the RL with the DSP. We expect that programming methodologies for the architecture will use a library-based design approach for the reconfigurable logic in which the designer simply describes the interconnection of the library modules using an assembly-like language. The intention is to make the programming task resemble more of a software creation problem than a hardware design exercise, a requirement crucial in making the architecture usable by DSP programmers and not just hardware designers. Future work will also quantify the effects of frequent reconfiguration on application performance.

References

[1] P. Graham and B. Nelson. FPGA-based sonar processing. In *Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '98)*, pages 201–208. ACM/SIGDA, ACM, 1998.

[2] P. Graham and B. Nelson. Frequency-domain sonar processing in FPGAs and DSPs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*. IEEE Computer Society, IEEE Computer Society Press, 1998.

[3] Rahul Razdan. *Programmable Reduced Instruction Set Computers.* PhD thesis, Harvard University, Cambridge,MA, May 1994.

[4] R. D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 126–135, Napa, CA, April 1996.

[5] Scott Hauck, Thomas W. Fry, Matthew M. Hosler, and Jeffery P. Kao. The Chimaera reconfigurable functional unit. In Kenneth L. Pocek and Jeffery Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96. IEEE Computer Society, IEEE Computer Society Press, April 1997.

[6] John R. Hauser and John Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, Napa, CA, April 1997.

[7] Charle R. Rupp, Mark Landguth, Tim Garverick, Edson Gomersall, Harry Holt, Jeffery M. Arnold, and Maya Gokhale. The NAPA adaptave processing architecture. In Kenneth L. Pocek and Jeffrey M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 28–37. IEEE Computer Society, IEEE Computer Society Press, April 1998.

[8] Jeffery A. Jacob and Paul Chow. Memory interfacing and instruction specification for reconfigurable processors. In *Proceedings of the Seventh ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '99)*, pages 145–154. ACM/SIGDA, ACM, 1999.

[9] Arthur Abnous and Jan Rabaey. Ultra-low-power domain-specific multimedia processors. In *Proceedings of the IEEE VLSI Signal Processing Workshop*. IEEE, IEEE, October 1996.

[10] Arthur Abnous, Katsunori Seno, Yuji Ichikawa, Marlene Wan, and Jan Rabaey. Evaluation of a low-power reconfigurable DSP architecture. In *Proceedings of the Reconfigurable Architectures Workshop*, March 1998.

[11] Scott Hauck. Configuration prefetch for single context reconfigurable coprocessors. In *Proceedings of the Sixth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '98)*, pages 65–74. ACM/SIGDA, ACM, 1999.

[12] Joe E. Brewer, L. Gray Miller, Ira H. Gilbert, Joseph F. Melia, and Doug Garde. A single-chip digital signal processing subsystem. In R. Mike Lea and Stuart Tewksbury, editors, *Proceedings of the Sixth Annual IEEE International Conference on Wafer Scale Integration*, pages 265–272, Piscataway, NJ, 1994. IEEE, IEEE.

[13] Csaba Andras Moritz, Donald Yeung, and Anant Agarwal. Exploring optimal cost-performance designs for Raw microprocessors. In Kenneth L. Pocek and Jeffery M. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '98)*, pages 12–27. IEEE Computer Society, IEEE Computer Society Press, 1998.

[14] Paul Graham and Brent Nelson. FPGAs and DSPs for sonar processing—inner loop computations. Technical Report CCL-1998-GN-1, Configurable Computating Laboratory, Electrical and Computer Engineering Department, Brigham Young University, 1998.

[15] Garrick Blalock. The BDTImark: A measure of DSP execution speed. Technical report, Berkeley Design Technology, Inc., 1997. Available at http://www.bdti.com/articles/wtpaper.htm.

[16] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the Seventh ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '99)*, pages 135–143. ACM/SIGDA, ACM, 1999.

[17] Alireza Kaviani, Daniel Vranesic, and Stephen Brown. Computational field programmable architecture. In *Proceedings for the IEEE Custom Integrated Circuits Conference (CICC '98)*, pages 12.2.1–12.2.4. IEEE, IEEE, 1998.

[18] André DeHon. *Reconfigurable Architectures for General-Purpose Computing*. PhD thesis, Massachusetts Institute of Technology, September 1996.

SLAAC: A Distributed Architecture for Adaptive Computing

Stephen P. Crago, Brian Schott, and Robert Parker
University of Southern California / Information Sciences Institute
4350 N. Fairfax Drive, Suite 770
Arlington, VA 22203
703-812-3729 (voice), 703-812-3712 (fax)

1. Introduction

Adaptive computing systems have successively demonstrated high performance on data and compute-intensive applications. Rencher and Hutchings showed that the Splash 2 executed a synthetic aperture radar automatic target recognition (SAR/ATR) algorithm 100 times faster than a conventional workstation [1]. Vuillemin *et al.* accomplished comparable speedups on the DECPeRLe-1 on a dozen different applications [2].

Although research efforts have proven that this class of architecture is effective, most existing adaptive computing systems suffer from hardware and software limitations that prevent an easy transition from the research lab to the field.

One of the primary architectural limitations shared by a majority of the existing adaptive computing systems is scalability. A considerable amount of research has been done defining novel architectures. However, these architectures have usually been limited to a single board or multiple boards connected to a common system bus. The problem is that bus-based systems do not scale well. These systems have high bandwidth within the board through custom interconnects and poor aggregate bandwidth among the boards on the system bus.

While adaptive computing systems have been shown to speed up many applications, they do not provide the best performance in all cases. Custom ASICs, when they exist for a given problem, can outperform adaptive computing systems. A well-balanced system might be comprised of a collection of boards based on a variety of technologies. Currently, there is little support for this heterogeneous environment.

In addition to the hardware limitations discussed above, a lack of code reuse is endemic to adaptive computing systems. Applications programmers generally cannot re-use code written for one adaptive computing system on a new system. Each system uses its own interface between the host and the nodes and between the application program and the runtime system. Adaptive computing systems are also lacking high-level compilers and programming tools. General-purpose computers were programmed in assembly language at one time, and adaptive computing systems are at a similar stage of development. High-level languages have been explored for adaptive computing systems [3], but are architecture specific and are not widely used.

Software tools, including debuggers and performance monitors, have been developed independently for specific adaptive computing systems. Consequently, a user has to learn a new set of tools when switching to a different architecture. Although at the level closest to the hardware the runtime system is necessarily different for specific systems, much of the runtime system software functionality is common across systems and could be standardized.

In this abstract, we propose the SLAAC (System Level Applications of Adaptive Computing) reference architecture to help address some of these issues in the adaptive computing community.

2. The SLAAC Reference Architecture

The purpose of the SLAAC Reference Architecture is to provide an open standard for the benefit of the adaptive computing community in order to facilitate collaboration among researchers and accelerate the transition of these technologies from the lab to the field. The SLAAC reference architecture does not define an adaptive computing system at the board level, but defines a common hardware and software interface to provide a framework for adaptive computing researchers to use. This common framework will allow common tools to be developed, common application interfaces to be used, and will provide a vehicle for deploying novel adaptive computing systems into the community.

2.1. Hardware Organization

As shown in Figure 1, the SLAAC system is organized as a network made up of hosts and semi-smart nodes. Hosts are defined as traditional computers that have an operating system and run user programs. The SLAAC reference architecture is heterogeneous. It allows for different types semi-smart nodes including compute devices such as FPGA (ACS) boards, and I/O devices such as disks and sensor interfaces.

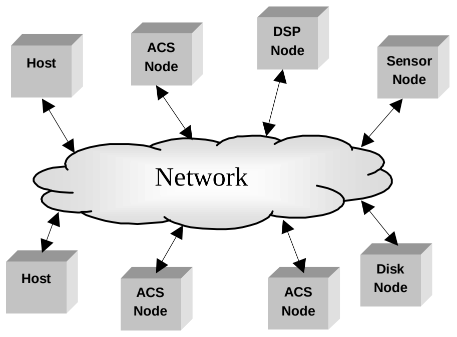


Figure 1: Network Distributed System

A node consists of a node controller and a compute or I/O device linked by a common bus. The node controller is responsible for making the device network capable. It manages network details such as addressing and data buffering. The node controller is also responsible for generating the control signals for the device on the node bus.

2.2. Software Layers

The SLAAC reference architecture defines a software interface that consists of several layers. Dividing the software interface into layers has two advantages. First, the lower layers can be used to hide device-dependent information from the programmer and make applications easier to program. Second, the layers will make it easier to integrate adaptive computing boards and tools into the system.

The three layers of the software interface are the control layer, the node layer, and the system layer. It is assumed that a method is provided to send messages over the network between hosts and nodes. The control layer is a set of commands generated by a host and interpreted by the node controller for interacting with the device at hardware level. The node layer provides a simplified programmer interface where the device-specific information is hidden. The system layer provides a means for allocating nodes and building larger systems using generic communication channels.

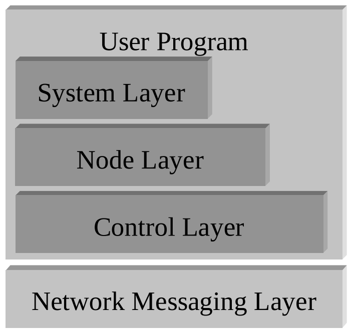


Figure 2: SLAAC Software Layers

3. Conclusion

The SLAAC architecture is not fixed. We are in the process of developing it and expect to go through many modifications as we continue to work closely with other research efforts. We would like to call for the input of applications programmers and adaptive computing researchers as to what interfaces are useful and likely to provide good performance.

4. Acknowledgements

Effort sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Material Command, USAF, under agreement number F30602-97-1-0222. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Rome Laboratory, or the U.S. Government."

[1] M. Rencher and B.L. Hutchings, “Automated Target Recognition on SPLASH 2,” *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1997, pp. 192-200.

[2] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, “Programmable Active Memories: Reconfigurable Systems Come of Age,” *IEEE Transactions on VLSI*, Vol.. 4, No. 1, March 1996.

[3] M. B. Gokhale and B. Schott, “A Data Parallel Programming Model,” In *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Press, 1996, pp. 77-96.

Using blocks of skewers for faster computation of Pixel Purity Index

James Theiler, Dominique D. Lavenier, Neal R. Harvey,
Simon J. Perkins, and John J. Szymanski

Space and Remote Sensing Sciences Group
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

ABSTRACT

The "pixel purity index" (PPI) algorithm proposed by Boardman, *et al.*[1] identifies potential endmember pixels in multispectral imagery. The algorithm generates a large number of "skewers" (unit vectors in random directions), and then computes the dot product of each skewer with each pixel. The PPI is incremented for those pixels associated with the extreme values of the dot products. A small number of pixels (a subset of those with the largest PPI values) are selected as "pure" and the rest of the pixels in the image are expressed as linear mixtures of these pure endmembers. This provides a convenient and physically-motivated decomposition of the image in terms of a relatively few components.

We report on a variant of the PPI algorithm in which blocks of B skewers are considered at a time. From the computation of B dot products, one can produce a much larger set of "derived" dot products that are associated with skewers that are linear combinations of the original B skewers. Since the derived dot products involve only scalar operations, instead of full vector dot products, they can be very cheaply computed.

We will also discuss a hardware implementation on a field programmable gate array (FPGA) processor both of the original PPI algorithm and of the block-skewer approach. We will furthermore discuss the use of fast PPI as a front-end to more sophisticated algorithms for selecting the actual endmembers.

Keywords: Hyperspectral, Multispectral, Endmembers, Remote Sensing, Pixel Purity, Field Programmable Gate Array (FPGA)

1. INTRODUCTION

A number of algorithms have been proposed over the last decade for finding so-called endmembers in multispectral data.[1–11] These algorithms are all based on the notion that a scene contains relatively few distinct materials and that much of the pixel-to-pixel variation in a multispectral image cube can be explained by the assumption that the pixels are "mixtures" of these distinct "pure" materials. The endmembers are the spectral signatures of these pure materials.

The "spectral unmixing" problem actually involves two steps. The first step is to find the endmember signatures (either from a library of known spectra or directly from the image), and the second is to express the individual pixels as (usually linear) combinations of these endmembers. The coefficients associated with each of the endmembers are identified with the "abundances" of the endmember materials in the given pixel; the coefficients are generally assumed to be proportional to the areal extent of each of the endmember materials in the pixel.

The emphasis of our work is on the computation involved in the first step, the identification of endmembers, directly from an image.

Like principal components, endmembers provide a basis set in terms of which the rest of the data can be described. But unlike principal components, the endmembers are expected to provide a more "physical" description of the data. There are two reasons for this expectation: One is that the endmembers are taken from the data themselves, and the other is that the linear combinations are restricted to combinations in which the coefficients are non-negative and sum to one. The mixed pixels can be literally interpreted as having X-percent of material A, and Y-percent of material B.

Emails: {`jt`,`lavenier`,`harve`,`s.perkins`,`szymanski`}`@lanl.gov`.

2. ORIGINAL PPI ALGORITHM

The importance of convex geometry as a paradigm for understanding the endmember problem was emphasized by Boardman *et al.*[1–3] early in this decade. If a multispectral image has D spectral channels, then each pixel can be identified with a point in a D-dimensional space.

Boardman's "pixel purity index" (PPI) algorithm[1] represents a specific attempt to exploit this paradigm, by locating data points which are on the vertices of the convex hull of this D-dimensional scatterplot. The problem of identifying the convex hull from a discrete set of data is a classic one in computational geometry, but the PPI algorithm provides a number of advantages over these classical algorithms. First, as we will describe below, it is relatively simple to implement, and it parallelizes quite naturally. Furthermore, it produces a relative measure for each vertex which distinguishes those near "corners" of the data with a higher-valued index: this is important because the corners are where the endmembers are expected to be. Although it is an approximate algorithm (it cannot be guaranteed to identify every vertex except in an asymptotic limit), it produces approximate answers right away, and the approximation gets better as the algorithm progresses. Finally, the penalty for working in a high-dimensional space increases roughly linearly (rather than exponentially) with the nominal dimension D of the space.

The algorithm proceeds by generating a large number N of random D-dimensional "skewers" through the D-dimensional data. For each skewer, every data point is projected onto the skewer, and the position along the skewer is is noted. The data points which correspond to extrema (or near extrema) in the direction of the skewer are identified, and placed on a list. As more skewers are generated, this list grows; the number of times a given pixel is placed on this list is also tallied. The pixels with the highest tallies are considered the most pure, and a pixel's count provides its "pixel purity index".

The PPI algorithm, by itself, does not identify a final list of endmembers. Taking the $D + 1$ "most pure" pixels, for instance, often leads to degenerate sets in which some of the endmembers are nearly identical. The pixel purity index was conceived not as a solution, but as a a guide; the author[2] proposed comparing the pure pixels with target spectra from a library, and successively projecting the data to lower dimensional spaces as endmembers were identified. Nonetheless, as Fig. 2 shows, the PPI algorithm can identify a small set of candidate endmembers from a large image. To illustrate the use of PPI and its variants, we use a 614×512-pixel 224-channel hyperspectral AVIRIS image[12]; we also use an image which is derived from the AVIRIS image, but only has ten channels, corresponding to bands available on the MTI satellite.[13]

Since PPI identifies more than just the $D+1$ points that make up the optimal simplex, it can potentially provide useful information to other algorithms (such as Archetypal Analysis[4] or N-FINDR[11]) to accelerate their convergence. This puts the PPI algorithm in the role of image pre-processor, and argues further in favor of its choice for hardware acceleration.

3. BLOCKS OF SKEWERS

The original PPI algorithm generates a large number of "skewers" (unit vectors in random directions), and then computes the dot product of each skewer with each data point. This can be computationally expensive, especially in high dimensions, and requires that a large number of dot products be evaluated.

Consider a block of B skewers, $\mathbf{k}_b \in \mathcal{R}^D$ with $b = 1, \ldots B$, and consider the B scalar dot products $d_b \in \mathcal{R}$ obtained from each of these skewers applied to one of the data points. The central observation in the blocks-of-skewers method is that for every skewer which is a linear combination of these B skewers, the "derived" dot product of that skewer with the data is a linear combination of the original dot products d_b. The reason this is useful is that derived dot products can be much cheaper to compute than original dot products. One price paid for this convenience is that the virtual skewers that produce the derived dot products are in the same B-dimensional subspace as the original B skewers. However, with appropriate choice of B and with a reasonable strategy for choosing the linear combinations, a considerable computational advantage can be obtained.

Given a set of B skewers, $\mathbf{k}_i \in \mathcal{R}^D$, for $i = 1, \ldots, B$; consider the (potentially much larger) set of virtual skewers given by the linear combinations $\mathbf{k}' = a_1\mathbf{k}_1 + \ldots + a_B\mathbf{k}_B$. There is in principle an infinite choice of coefficients a_i, $i = 1, \ldots, B$, but we will consider various restrictions below. But even for arbitrary $a_i \in \mathcal{R}$, already there is a computational advantage that can be seen.

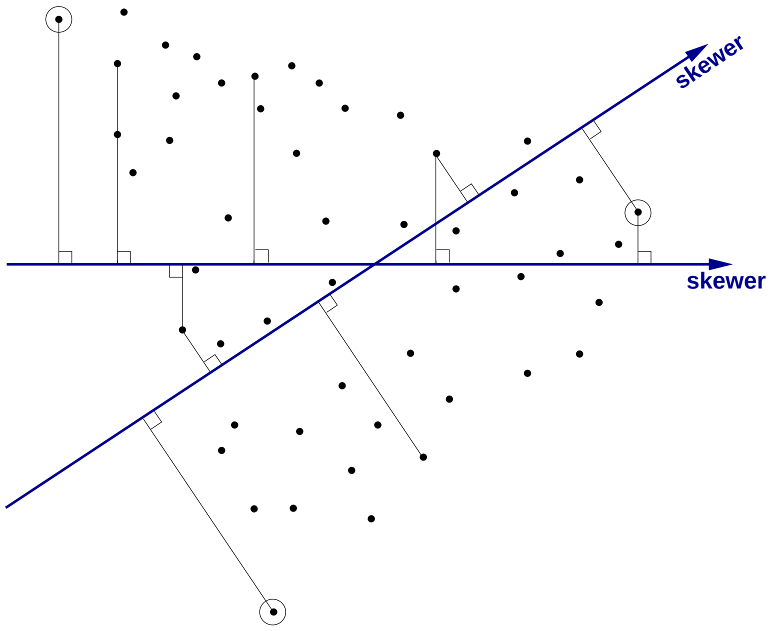


Figure 1. The PPI algorithm works by projecting points in the data set onto random skewers. For each skewer, two extreme points are identified, and their pixel purity index is incremented. In the figure above, the circled points are identified as candidate endmembers in the full space because their projection onto one or both of the skewers is extremal.

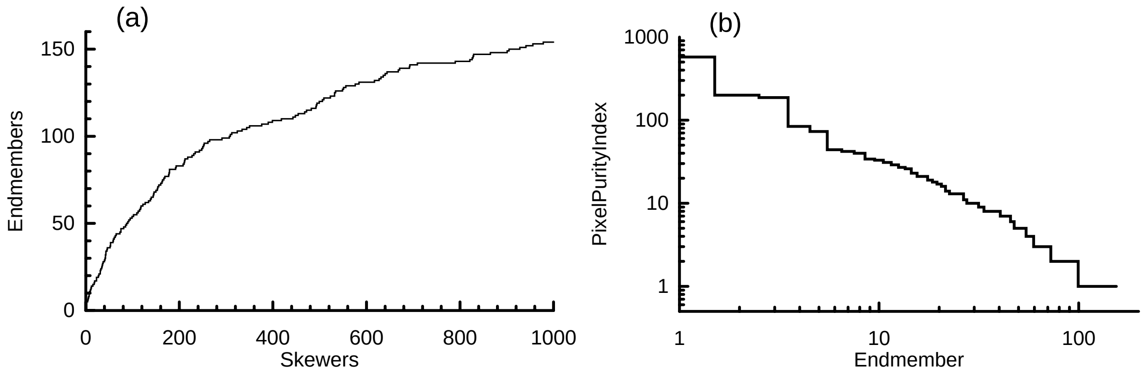


Figure 2. **(a)** Plot of the cumulative number of endmembers found as a function of the number of skewers in a run of PPI on a ten-channel multispectral data set. After the run of $N = 1000$ skewers, a total of 154 candidate endmembers were produced. **(b)** Sorting the candidate endmembers according to their pixel purity index shows the range of pixel purity indices exhibited by the candidate endmembers: a few have very large values, some have intermediate values, and fully a third of them have an index of unity. Those points with higher indices are presumed to be closer to the "corners" of the data in the ten-dimensional space.

(a)

$$\begin{array}{ccccc}
p_1 & & & & \\
 & & p_2 & & \\
p_1 & + & p_2 & & \\
p_1 & - & p_2 & & \\
 & & & & p_3 \\
p_1 & & & + & p_3 \\
 & & p_2 & + & p_3 \\
p_1 & + & p_2 & + & p_3 \\
p_1 & - & p_2 & + & p_3 \\
p_1 & & & - & p_3 \\
 & & p_2 & - & p_3 \\
p_1 & + & p_2 & - & p_3 \\
p_1 & - & p_2 & - & p_3 \\
 & & \vdots & &
\end{array}$$

(b)

$$\begin{array}{ccccccc}
p_1 & + & p_2 & + & p_3 & + & \cdots \\
p_1 & - & p_2 & + & p_3 & + & \cdots \\
p_1 & - & p_2 & - & p_3 & + & \cdots \\
p_1 & + & p_2 & - & p_3 & + & \cdots \\
p_1 & + & p_2 & - & p_3 & - & \cdots \\
p_1 & - & p_2 & - & p_3 & - & \cdots \\
p_1 & - & p_2 & + & p_3 & - & \cdots \\
p_1 & + & p_2 & + & p_3 & - & \cdots \\
 & & & \vdots & & &
\end{array}$$

Figure 3. If $p_1, p_2, p_3, \ldots$ are the original dot products, computed from a block of skewers applied to a single data point, then the derived dot products are linear combinations of these: $a_1p_2 + a_2p_2 + a_3p_3 + \cdots$. Appropriately ordering the derived dot product calculations can lead to considerably reduced computation at each step. **(a)** For the simple discrete case in which $a_i \in \{-1, 0, 1\}$, each of the derived dot products associated with the $O(3^B)$ virtual skewers are obtained by a single addition or subtraction to a term that has been computed previously. **(b)** For the corners of the hypercube case, with $a_i \in \{-1, 0, 1\}$, an arrangement like a graycode leads to a set of computation in which each term is equal to the previous term plus or minus twice one of the original dot products.

For each of the D-dimensional data points $\mathbf{x_n}$, the B dot products $\mathbf{k}_i \cdot \mathbf{x_n}$, for $i = 1, \ldots, B$. each require D multiply-accumulate operations. But the derived dot product

$$\mathbf{k}' \cdot \mathbf{x_n} = a_1\mathbf{k}_1 \cdot \mathbf{x_n} + \ldots + a_B\mathbf{k}_B \cdot \mathbf{x_n} \tag{1}$$

can be expressed as a linear combination of these B dot products, and can be computed with only B multiply-accumulates. As long as $B < D$, these derived dot products are cheaper to compute than the original dot products; on the other hand, the derived dot products are with vectors that are in the B-dimensional subspace spanned by $\mathbf{k}_i$ for $i = 1, \ldots, B$.

3.1. Discrete Linear Combinations

If, rather than permit arbitrary coefficients, we restrict them to be small discrete integers, *i.e.*, $a_i \in \{-n, -(n-1), \ldots, -1, 0, 1, \ldots, n-1, n\}$, then multiplication by a_i can be implemented as a small number of additions. In this case, then there are $(2n+1)^B$ of these virtual skewers, though the useful count is $((2n+1)^B - 1)/2$, neglecting the $\mathbf{k}' = 0$ skewer and recognizing that the overall sign of the skewer doesn't matter (since we consider both maximum and minimum values of the dot products).

Here, even $n = 1$ provides an $O(3^B)$ gain, though not all of this exponential gain can be realized in practice. The calculation of each derived dot product nominally requires $O(B)$ additions, though by computing the derived dot products in a well-chosen order, this number can be reduced to $O(1)$; see Fig. 3(a). The price per dot product can be greatly reduced, but the dot products must still be applied to every data point. And for each skewer, virtual or otherwise, one must keep track of maximum and minimum dot product, as the algorithm is looped over the entire data set.

The speedup that is available from these virtual skewers is considerable. For a high dimensional space, it is the multiply-accumulates in the original dot products that will dominate the computation, and since we get $(3^B - 1)/2$

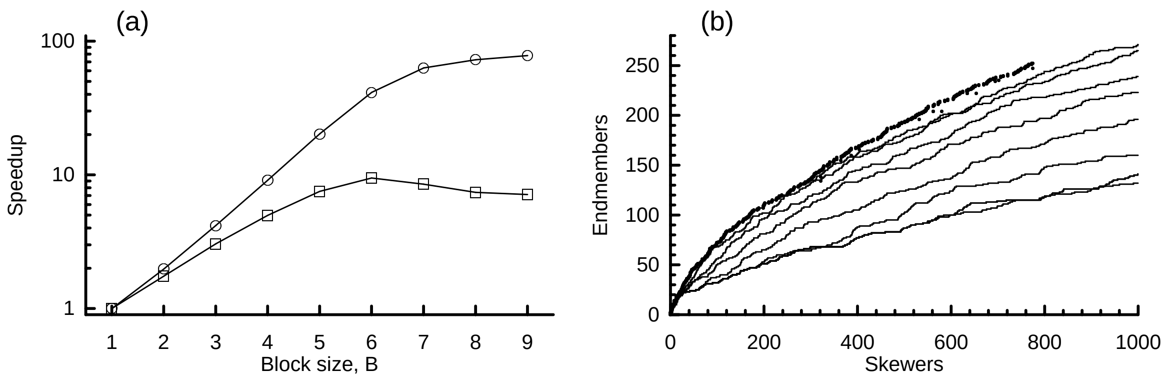


Figure 4. **(a)** Speedup available from the block-skewer method increases rapidly with block size B, but saturates at a speedup that scales with the dimension D of the data. The circles represent the speedup attained when the block-skewer method was applied to find endmembers of a 224-channel AVIRIS image; the squares are the same calculation, but applied to a 10-channel simulated MTI image. In both cases, discrete linear combinations with $a_i \in \{-1, 0, 1\}$ were used. **(b)** A price is paid for this speedup. Shown here are the cumulative endmembers found as a function of number of skewers. The darker line corresponds to the standard (slow) $B = 1$ algorithm, and the performance generally decreases with increasing B.

virtual skewers out of our B original skewers, the theoretical speedup is

$$S_o = \frac{3^B - 1}{2B}. \tag{2}$$

However, if we account for the time required for an addition or subtraction ($t_{\mathrm{add/sub}}$), which is needed for every virtual skewer, the time required to maintain a record of the minimum and maximum dot product so far ($t_{\mathrm{min/max}}$), then we obtain a formula for speedup

$$S = \frac{S_o}{1 + S_o\tau/D}, \tag{3}$$

where D is the number of spectral channels, and τ is the ratio

$$\tau = \frac{t_{\mathrm{add/sub}} + t_{\mathrm{min/max}}}{t_{\mathrm{mult/acc}}}. \tag{4}$$

For large D, this looks like the idealized speedup S_o until it saturates at $S_{\mathrm{top}} = 1/(D\tau)$.

3.2. Corners of the hypercube

If we further restrict consideration to virtual skewers $\mathbf{k}' = \sum_{i=1}^{B} a_i\mathbf{k}_i$, with $a_i \in \{-1, 1\}$, then there are 2^{B-1} derived dot products for every B original dot products. This is not as large as the $O(3^B)$ gain seen in the previous section, but the gain is still exponential.

The advantage comes in comparing the angular separation of nearby skewers. If $\mathbf{k}_1$ and $\mathbf{k}_2$ are two skewers, then

$$\cos\theta = \frac{\mathbf{k}_1' \cdot \mathbf{k}_2'}{|k_1'|\,|k_2'|} \tag{5}$$

defines the angle between them. The closest pairs of skewers will be those whose a_i values differ only at a single i. For those pairs, assuming the original B skewers are orthogonal, the angle between them will be given by

$$\cos\theta = 1 - \frac{2}{B} \tag{6}$$

By contrast the closest pair of skewers when a_i is permitted to include 0 as well as $\{-1, 1\}$ is given by

$$\cos\theta = \frac{B-1}{\sqrt{(B-1)B}} \approx 1 - \frac{1}{2B}. \tag{7}$$

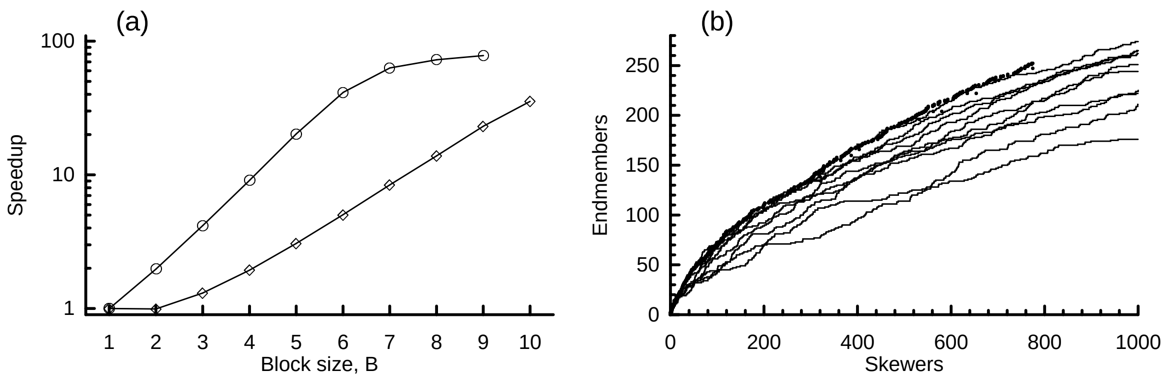


Figure 5. **(a)** Speedup for two different block-skewer methods; the first (circles) is the discrete linear combinations with $a_i \in \{-1, 0, 1\}$ were used, which is the same as shown in Fig. 4(a). The second is the alternate corners method, with $a_i \in \{-1, 1\}$, and only every other corner used. The speedup of the alternate corners method does not increases as rapidly with block size B, but it also does not saturate for as small a value of B. **(b)** The real advantage of the alternate corners method, compared to all discrete linear combinations with $a_i \in \{-1, 0, 1\}$, is seen in the plot of cumulative endmembers found as a function of number of skewers. Comparing this figure to Fig. 4(b) shows that more endmember candidates are found.

There are *more* virtual skewers per block when zero-valued coefficients are permitted, but those skewers are also closer to each other.

Restricting ourselves to corners of the hypercube, we get fewer virtual skewers per block ($O(2^B)$ instead of $O(3^B)$), but those skewers are more widely separated. For the same number of virtual skewers, we can use a larger block size B, and therefore cover a higher dimensional subspace of the full D-dimensional sphere with a single block.

The first derived dot product requires $B - 1$ additions (but with $B \ll D$, this is a lot cheaper than the D multiply-accumulate's required for a real dot product); furthermore, by organizing the order of computation of the derived skewers like a "graycode," it is possible to compute each of the remaining derived dot products with only a single addition; see Fig. 3(b).

3.3. Alternate corners

It is clear from the previous section that a goal in choosing directions for virtual skewers is to try and cover as much of the surface of the D dimensional unit sphere as possible. The alternate corners strategy considers a subset of the 2^B coefficient vectors $\mathbf{a} \in \{-1, 1\}^B$ which still "covers" the B dimensional space, but not as finely as the full set. One measure of the granularity of this coverage is the characteristic distance between coefficent vectors and their nearest neighbors in the B-dimensional coefficient vector space. For binary-valued coefficients, this is effectively a Hamming distance; and when the Hamming distance between a pair of $\mathbf{a}$'s is k, then the angle between the derived skewers is θ where

$$\cos\theta = 1 - \frac{2k}{B} \tag{8}$$

The case $k = 1$ produces all the corners of the hypercube; $k = 2$ produces only alternate corners, of which there are 2^{B-1} in a B-dimensional cube. For even-sized blocks $B \equiv 0 \pmod 2$, the corners on opposite ends of the cube are alternate corners; if we compute both min and max of every derived dot product, then the alternate corners will be taken care of, and we will effectively get 2^{B-2} derived dot products for each block of B skewers. (For odd-sized blocks $B \equiv 1 \pmod 2$, there is no advantage to taking alternate corners since computing both min and max of every dot product is effectively the same as using all corners.)

The advantage of having virtual skewers farther apart is illustrated in Fig. 5. The speedup increases less rapidly with B, but for a given speedup, more endmembers are identified.

In general, the problem of identifying a maximal set of vectors in a B-dimensional binary space such that every pair of vectors has a Hamming distance at least as large as a specified k is a classic problem in communication theory.

One would like to produce "codes" for which few-bit errors and be detected and/or corrected, and to do so requires sets of binary vectors with large pairwise Hamming distances. A "perfect" code satisfies

$$A(B,k) = \frac{2^B}{\sum_{i=0}^{(k-1)/2} \binom{B}{i} 2^i}, \tag{9}$$

but for for most values of B and k, this is an upper bound. In fact, for many values of B and k, the actual size of the maximal code is unknown. Nonetheless, algorithms exist for finding "good" codes, given values of B and k. For $k = 3$, Hamming found perfect codes whenever B is of the form $2^m - 1$; the number of code vectors in this case is given by 2^{2^m-1-m}, and so the gain is given by

$$\frac{2^{2^m-1-m}}{2^m - 1} \approx \frac{2^B}{B^2} \tag{10}$$

virtual skewers per actual skewer. (See Roman[14] for an accessible introduction to error-correction and error-detection coding.)

3.4. Orthogonally aligned skewers

In previous sections we have spoken of randomly chosen vectors $\mathbf{k}_b \in \mathcal{R}^D$; however, we can avoid even computing the original B dot products by choosing the vectors $\mathbf{k}$ appropriately. In particular, we consider choosing the $\mathbf{k}$'s from among the D axes of the original data. There are

$$\left(\begin{array}{c} D \\ B \end{array} \right) = \frac{D!}{B!(B-D)!} \tag{11}$$

ways to choose a block of B random othogonal skewers from the D orthogonal components, so for even moderately large D and B, we are not giving up a lot of randomness by limiting ourselves to orthogonal vectors.

3.5. Principal Components

An advantage of the PPI algorithm over some alternatives is that it works reasonably well in high-dimensional spaces; if the effective dimension of a data set is D', but the data are nominally expressed as D channels, then the only penalty for working in the nominal space instead of the reduced space is D/D'. For some algorithms, this penalty is exponential in D.

Nonetheless, there is still an advantage in working in a reduced space if one is available. And principal components analysis provides a relatively convenient way of projecting the data to a lower dimensional space while still maintaining most of the variability in the data. The principal components algorithm is not cheap, however, and the matrix factorization involved is very floating-point intensive, which makes a full-hardware implementation less attractive.

The first (and most expensive) step in a principal components analysis is the computation of the covariance matrix. This requires $O(ND^2)$ scalar multiply-accumulate operations, although shortcuts are feasable here (*e.g.*, use a sampling of $N' \ll N$ points to estimate the covariance matrix; or use the covariance matrix from a previous experiment). The matrix factorization, usually a singular value decomposition, does not have the O(N) pre-factor, but it does scale as $O(D^3)$, with a large (of order ten) coefficient. Having determined the D' largest principal components, one can rotate the data with $O(ND')$ dot products, each in the nominal D-dimensional space, for a total effort of $O(NDD')$ multiply-accumulates.

The standard PPI algorithm, with K skewers, requires $O(NDK)$ multiply-accumulates in the nominal space, and $O(ND'K)$ in the lower-dimensional space.

With $K \gg D$, the cost of rotating the data is small compared to the cost of the PPI, so the gain in going to principal components is the factor D'/D. This is only linear in the dimension, but for hyperspectral data with hundreds of channels, and projections to a few tens of principal components, gains of an order of magnitude are available.

A possibly greater gain, in combining principal components with the fast PPI algorithm, lies in the projection of the D-dimensional space to a lower B-dimensional space.

4. HARDWARE ACCELERATION

The pixel purity algorithm is a good candidate for acceleration because it is readily parallelizable (the skewer calculations are done independently) and the core calculation is a simple dot product. Hardware acceleration of a dot product is by itself useful, as it is the "inner loop" for many remote sensing and multispectral image processing tasks.[15]

This includes the N-FINDR[11] algorithm, which attempts to inscribe the largest-volume simplex in the data cloud. Here, the quality of a given data point as the potential replacement for a given fiducial vertex can be expressed as a dot product of that data point with the "skewer" that is perpendicular to the sub-simplex that does not contain the fiducial vertex. The computation of the skewer is somwhat involved, but it scales only with the dimension of the simplex, and not with the size of the data set. So for large data sets, the dominant computation is the dot product of very particular skewers with the full dataset.

We have designed an implementation of the standard PPI algorithm on reconfigurable hardware.[16] The basic architecture is shown in Fig. 6. It is composed of a matrix of dot-product operators. Each dot-product is fed serially with the D components of the pixels and the skewers; this requires D cycles, but in that time a dot product is computed for each operator in the matrix.

The results of the dot-product are stored into registers (shaded boxes) and shifted to the MinMax units. These units compute the minimum and the maximum of the dot-product and keep track of which pixel produced the extreme values. The results of the MinMax units are shifted to the host processor through a fifo.

In order to get the best performance, the minimum and maximum operations are performed in parallel with the dot-product computation: as soon as a dot-product phase is accomplished, the results are stored in the shift registers, and another phase begins immediately. During the next dot-product phase computation, the previous dot-product values are bit-serially shifted to the MinMax units. Hence, there is a complete overlap between the dot-product and the minimum/maximum computations.

One issue that arises in the hardware implementation is the precision of skewer coefficients. If the skewers are limited to a few bits of precision per coordinate direction, they can still provide an effective "cover" of the D-dimensional sphere of possible directions, but the dot product computation is faster (and more to the point, since it takes up less real estate on the reconfigurable chip, is more parallelizable). In our implementation, we use three bits (allowing the coefficient to vary from -3 to 3), but Fig. 7 shows the effect of reduced precision in the plot of cumulative endmember candidates. Reduced precision does indeed reduce the number of candidates, but the effect is small, which gives us confidence in this design choice.

We have started to investigate the hardware acceleration of the block-skewer approach, but this is currently a work in progress.

5. PPI AS A PRE-PROCESSOR

Since PPI produces candidate endmembers, we can use these as input to more sophisticated algorithms which can then be used to identify the best choice of endmembers. To illustrate the idea, we used PPI to produce a stream of endmember candidates, and then used these pixels as input to our implementation of the NFINDR[11] algorithm. NFINDR attempts to inscribe the largest volume simplex in the data and the effort to do so scales linearly with the data. (It is to the credit of the NFINDR authors that it is so fast – there is a combinatorially large number of possible simplices one might inscribe in a given data set.) Although NFINDR is quite efficient at processing data, some more sophisticated approaches, such as the archetypes of Cutler and Brieman,[4] require processing that scales very rapidly with the number of data points. If a fast PPI can be used to rapidly identify candidate endmembers, then these more sophisticated approaches can use that to speed up their efforts to identify the actual endmembers. In Fig. 8, we supply NFINDR with endmember candidates from PPI applied to ten-dimensional simulated MTI data, and we find that it is able to rapidly identify the largest volume simplex from this data. As a comparison, we supply NFINDR with random subsets of the pixels from the image, and again it finds the largest volume simplices that these subsets can support. What is most evident in this comparison is that the simplex volume computed with only a few of the PPI candidate endmembers is much larger than the volume computed with random subsets of much greater size.

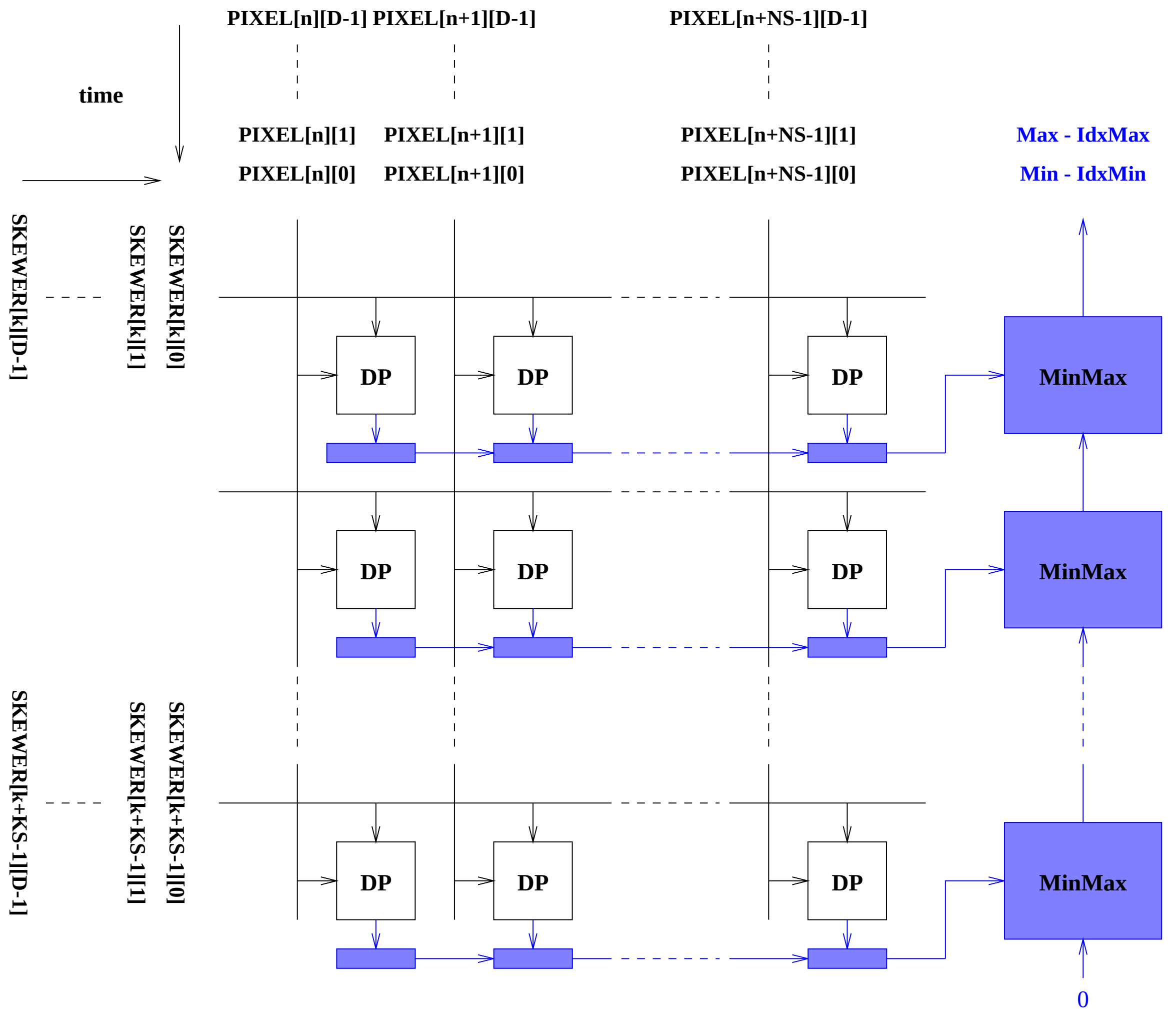


Figure 6. Basic architecture of the hardware implementation of the Pixel Purity Index algorithm. A matrix of dot-product operators (DP) operate in parallel, taking the components of the skewers on the horizontal axis and the components of the pixels on the vertical axis. The results are fed into MinMax units which identify the extreme dot products.

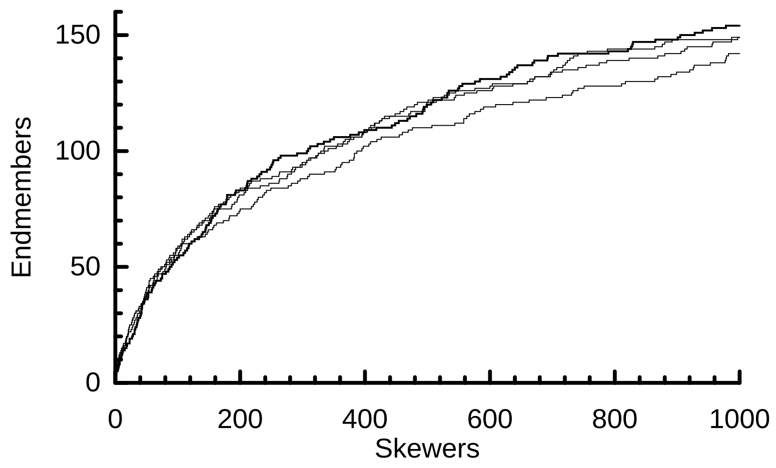


Figure 7. Cumulative plot of candidate endmembers as a function of the number of skewers is shown for different discretizations of the skewer coefficients. The darker line at the top is the standard PPI, and is the same as Fig. 2(a); the lighter lines correspond to skewer coefficients discretized to 2, 3, and 4 bits of precision. For example, the 3-bits case only permits the seven integer coefficients that vary from -3 to 3, inclusive. The 2-bits case is the lower curve, and in general, the more bits of precision that are used, the more candidate endmembers are found, but beyond two bits, the effect is not substantial.

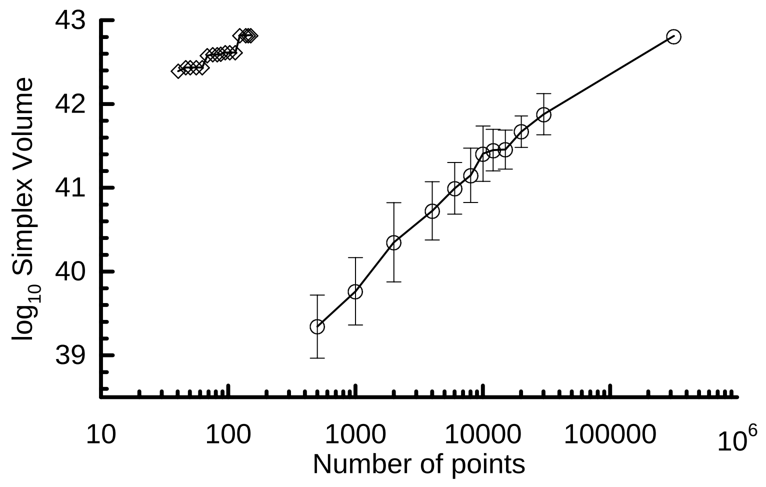


Figure 8. This plot shows the results of two separate experiments. In both experiments a number of pixels was supplied to NFINDR, and NFINDR produced the volume of the largest simplex supported by the pixels. The diamonds correspond to the first 150 candidate endmembers identified by PPI. The circles correspond to large but random subsets of the pixels in the image. This figure illustrates the effectiveness of PPI as a preprocessor.

ACKNOWLEDGEMENTS

We are pleased to acknowledge Chris Borel, for pointers to the literature on pixel purity, and for general discussions of endmembers and their applications. We also thank the Multispectral Thermal Imager team for supplying AVIRIS and simulated MTI data. Finally, we are grateful to the other members of the ASAPP team for many useful discussions.

REFERENCES

1. J. W. Boardman, F. A. Kruse, and R. O. Green, "Mapping target signatures via partial unmixing of AVIRIS data," in *Summaries of Fifth Annual JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 23–26, 1995.
2. J. W. Boardman, "Automating spectral unmixing of AVIRIS data using convex geometry concepts," in *Summaries of the Fourth Annual JPL Airborne Geoscience Workshop*, R. O. Green, ed., pp. 11–14, 1994.
3. J. W. Boardman, "Geometric mixture analysis of imaging spectrometry data," *IEEE* , pp. 2369–2371, 1994.
4. A. Cutler and L. Breiman, "Archetypal analysis," *Technometrics* **36**, pp. 338–347, 1994.
5. D. A. Roberts, M. Gardner, R. Church, S. Ustin, G. Scheer, and R. O. Green, "Mapping chaparral in the Santa Monica mountains using multiple spectral mixture models," in *Summaries of 6th JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 197–201, 1996.
6. S. Tompkins, J. F. Mustard, C. M. Pieters, and D. W. Forsyth, "Optimization of endmembers for spectral mixture analysis," *Remote Sensing of the Environment* **59**, pp. 472–489, 1997.
7. G. S. Okin, W. J. Okin, D. A. Roberts, and B. Murray, "Multiple endmember spectral mixture analysis: Application to an arid/semi-arid landscape," in *Summaries of the Seventh JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 291–299, 1998.
8. C. A. Bateson, G. P. Asner, and C. A. Wessman, "Incorporating endmember variability into spectral mixture analysis through endmember bundles," in *Summaries of the Seventh JPL Airborne Earth Science Workshop*, R. O. Green, ed., pp. 43–52, 1998.
9. J. Bowles, M. Daniel, J. Grossman, J. Antoniades, M. Baumback, and P. Palmadesso, "Comparison of output from ORASIS and pixel purity calculations," *Proc. SPIE* **3438**, pp. 148–156, 1998.
10. A. Mooijaart, P. G. M. van der Heijden, and L. A. van der Ark, "A least squares algorithm for a mixture model for compositional data," *Computational Statistics and Data Analysis* **30**, pp. 359–379, 1999.
11. M. E. Winter, "N-FINDR: an algorithm for fast autonomous spectral end-member determination in hyperspectral data," *Proc. SPIE* **3753**, pp. 266–277, 1999.
12. NASA, 1999. `http://makalu.jpl.nasa.gov/avaris.html`.
13. P. G. Weber, B. C. Brock, A. J. Garrett, B. W. Smith, C. C. Borel, W. B. Clodius, S. C. Bender, R. R. Kay, and M. L. Decker, "MTI Mission Overview," *Proc. SPIE* **3753**, pp. 340–346, 1999.
14. S. Roman, *Introduction to Coding and Information Theory*, Springer, New York, 1997.
15. M. P. Caffrey, A. Begtrup, J. Layne, T. Nelson, S. Robinson, A. Salazar, J. J. Szymanski, and J. Theiler, "High performance signal and image processing for remote sensing using reconfigurable computers," *Proc. SPIE* **3807**, 1999.
16. D. Lavenier and J. Theiler, "FPGA implementation of the Pixel Purity Index algorithm for hyperspectral images," Tech. Rep. LA-UR-00-2426, Los Alamos National Laboratory, 2000.